VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Analýza lidského genomu

Human Genome Analysis

2019 Bc. Jan Kratochvíl

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jan Kratochvíl**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Analýza lidského genomu
Human Genome Analysis

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je navrhnout a implementovat aplikaci umožňující analýzu lidského genomu. Přepokládá se napojení na existující výpočetní řetězec, návrh možných vylepšení s návazností na další analytické či statistické nástroje (např. R).

Body zadání:
1. Přehled aktuálních trendů v genovém sekvenování a mapování.
2. Zmapování výpočetního řetězce, identifikace slabých míst.
2. Návrh možných vylepšení s využitím paralelních výpočtů, analytických či statistických nástrojů, apod.
3. Návrh a implementace aplikace.
4. Vytvoření experimentů, zhodnocení dosažených výsledků a cílů práce.

Seznam doporučené odborné literatury:

[1] M. Pilgrim: HTML5: Up and Running, O'Reilly Media; 1 edition (August 24, 2010), ISBN-13: 978-0596806026
[2] D. Crockford: JavaScript: The Good Parts, O'Reilly Media; 1st edition (May 2008), ISBN-13: 978-0596517748

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Ing. Petr Gajdoš, Ph.D.**

Datum zadání: 01.09.2018
Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2019 .....

Tímto chci poděkovat doc. Ing. Petru Gajdošovi, Ph.D. za jeho odborné vedení při tvorbě této práce. Dále chci poděkovat své rodině za podporu při studiu.

Abstrakt

Tato diplomová práce se zabývá implementací sufixových automatů, které jsou využity ve vyhledávání řetězců v DNA sekvencích. V první části práce je seznámení s problematikou sekvenování a mapování DNA. Následuje teoretická část popisující datové struktury sufixový strom a sufixové pole využívané ve vyhledávání v textu. Dále je seznámení se sufixovými automaty, na které navazují kompaktní sufixové automaty, návrh a implementace této struktury. Implementace je zaměřena na rozdělení vstupního řetězce na několik podřetězců, kde pro každý tento podřetězec je sestrojen sufixový automat. Bylo provedeno několik experimentů nad implementací této datové struktury. Výsledky experimentů jsou shrnuty v závěru této práce.

Klíčová slova: DNA sekvenování, sufixový strom, sufixové pole, sufixový automat, DAWG, kompaktní sufixový automat, CDAWG

Abstract

This thesis describes the implementation of suffix automatons used for string searching on long DNA sequences. The first chapter talks about DNA sequencing and mapping. Then follows a theoretic primer on the topic of suffix trees and suffix arrays which are widely used for searching over long strings. The next chapter introduces suffix automatons, which are followed by compact suffix automatons, design draft and implementation of this structure. The implementation focuses on splitting the input string into several substrings, where for each substring a suffix automaton is constructed. A wide number of experiments have been conducted over this data structure. Finally, the results from various experiments are summed up in the closing section.

Key Words: DNA sequencing, suffix tree, suffix array, suffix automaton, DAWG, compact suffix automaton, CDAWG

Obsah

Seznam použitých zkratek a symbolů **7**

Seznam obrázků **8**

Seznam tabulek **9**

1 Úvod **11**

2 Analýza DNA **12**
2.1 Sekvenování genomu 12
2.2 Genové mapování 14
2.3 Projekt Lidského Genomu 15

3 Využívané datové struktury **16**
3.1 Sufixový strom 16
3.2 Sufixové pole 19

4 Návrh a implementace sufixových automatů **21**
4.1 Sufixový automat 21
4.2 Kompaktní sufixový automat 26
4.3 Implementace 29
4.4 Zápis na disk 32
4.5 *Lazy-loading* 33
4.6 Komprese dat 34
4.7 Vyhledávání řetězců 35
4.8 Možnosti paralelního zpracování 40

5 Experimenty **42**
5.1 Vstupní data 42
5.2 Čas pro konstrukci automatů 43
5.3 Velikosti automatů 44
5.4 Průměrný čas vyhledání řetězce 44
5.5 Vyhledávání v jediném automatu 46
5.6 *Lazy-loading* při vyhledávání 47
5.7 Vyhledávání delších řetězců 47
5.8 Nalezení prvního výskytu řetězce 48
5.9 Experimenty na *human* datech 49

6 Závěr **51**

Seznam použitých zkratek a symbolů

ST	–	Suffix Tree
STrie	–	Suffix Trie
SA	–	Suffix Array
DAWG	–	Directed Acyclic Word Graph
CDAWG	–	Compact Directed Acyclic Word Graph
DNA	–	Deoxyribonucleic acid
A	–	Adenin
C	–	Cytosin
T	–	Thymin
G	–	Guanin
HGP	–	Human Genome Project
HUGO	–	Human Genome Organization

Seznam obrázků

1 Schéma sekvenování Sangerovou metodou. 13
2 Vizualizace elektorferogramu při sekvenování DNA Sangerovou metodou. 13
3 Převod mezi datovými strukturami 16
4 Sufixová trie $STrie(abaababa)$ a sufixový strom $ST(abaababa)$. 17
5 Kroky konstrukce sufixového stromu $ST(cacao)$. 19
6 Konstrukce sufixového automatu $DAWG(ababcb)$ ze sufixové trie. 22
7 Případ minimálního automatu, který není platným $DAWG$. 23
8 Strom sufixových odkazů pro $DAWG(ababcb)$. 24
9 Jednotlivé kroky přímé konstrukce $DAWG(ababcb)$ 26
10 Chybně zkonstruovaný $DAWG$. 27
11 Minimalizací sufixového stromu vznikne $CDAWG$. 27
12 Zpětný převod $CDAWG$ do $DAWG$. 28
13 Přímá konstrukce kompaktního sufixového automatu $CDAWG(ababcb)$. 30
14 Rozdělení na podřetězce při konstrukci $CDAWG$. 31
15 Ilustrace pomocné indexové struktury pro *lazy-loading*. 34
16 $CDAWG(ababcbabcbabbaacb)$ 37
17 $CDAWG(ababcbab)$ 39
18 Schéma návrhu paralelní konstrukce automatů 40
19 Schéma návrhu paralelního vyhledávání 41
20 Graf porovnávající celkový čas konstrukce všech automatů 43
21 Srovnání velikostí automatů na disku. 44
22 Grafy průměrných časů s rozdělením od 2^5 do 2^{15} znaků. 45
23 Graf průměrných časů pro 3 sady dotazů. 46
24 Srovnání času konstrukce a vyhledání v jednom automatu. 46
25 Graf velikostí automatů na disku při rozdělení 2^{15}, 2^{17} znaků a bez rozdělení (jeden automat). 47
26 Graf rychlosti vyhledávání s využitím *lazy-loading*. 48
27 Srovnání času vyhledávání delších řetězců. 48
28 Srovnání času vyhledávání prvních výskytů řetězců. 49

Seznam tabulek

1 Sufixové pole pro $S = banana$ 20
2 Tabulka hodnot pro znaky abecedy při kompresi 34
3 Vyhledávací tabulka pro kompresi 35
4 Tabulka indexů pro řetězec $S = ababcbabcbabbaacb$ 37
5 Tabulka přechodů při vyhledávání řetězce $X = cbab$. 37
6 Tabulka přechodů při vyhledávání řetězce $X = c$. 38
7 Tabulka rozdělení řetězce na 3 části. 39
8 Tabulka přechodu při vyhledávání řetězce $X = abcabc$. 39
9 Příklad vygenerované sady dotazů 43
10 Vstupní soubory pro experimenty 43
11 Experimenty na *human* datech. 49

Výpis algoritmů

1 Algoritmus přímé konstrukce $DAWG$. 25
2 Algoritmus konstrukce $CDAWG$ kompresí $DAWG$. 28
3 Algoritmus přímé konstrukce $CDAWG$ v lineárním čase. 29
4 Datové struktury pro hrany a stavy v $CDAWG$. 32
5 Algoritmus zápisu $CDAWG$ na disk 33
6 Algoritmus nalezení všech výskytů řetězce 36

1 Úvod

Efektivní vyhledávání v textu je klíčové pro aplikace v oblasti bioinformatiky, analýzu časových řad, textové editory nebo kompresi dat. Přestože se výkon výpočetních zařízení neustále zvyšuje, zároveň roste objem dat, která je potřeba zpracovávat, jako jsou například biologické sekvence. Je tedy nezbytné vyvíjet datové struktury pro efektivní uložení dat, techniky pro dotazování nad daty či analýzu těchto dat. Sufixový strom, sufixová pole a sufixové automaty jsou struktury, které se typicky používají pro vyhledávání nad textem.

V oblasti bioinformatiky při zpracování DNA sekvencí se pracuje s gigabajty dat. Pro sufixové stromy a sufixová pole může takové množství informací představovat problémy. V této práci jsou však zmíněny algoritmy, které dokáží tyto komplikace obejít. Sufixové stromy obsahují velké množství nadbytečných informací. Řešením je využít sufixové automaty $DAWG$ a kompaktní sufixové automaty $CDAWG$, které dokáží efektivněji pracovat s pamětí oproti stromům. Jsou to deterministické konečné automaty, které přijímají všechny sufixy řetězce, nad kterým jsou sestaveny. Protože přijímají všechny sufixy, lze je využít i pro vyhledávání uvnitř řetězců.

Tato práce se zaměřuje na sufixové automaty $DAWG$, zejména pak na kompaktní sufixové automaty $CDAWG$. Jsou popsány metody konstrukce této struktury, implementace, její výhody a porovnání se stromovými strukturami. Experimenty nad touto strukturou jsou založeny na lokalizaci vyhledávání, kde je nad vstupem sestrojeno několik sufixových automatů. Pokud jsou známy pozice ve vstupním řetězci, kde chceme vyhledávat, pozice hledaných řetězců lze dohledat velice efektivně. V práci je popsán návrh takové implementace a její realizace, způsob zápisu dat na disk, metody komprese dat. Jsou diskutovány možnosti paralelního zpracování této datové struktury a způsob vyhledávání řetězců. Následují praktické experimenty nad touto strukturou, které se zabývají délkou konstrukce této struktury, velikostí struktury při zápisu na disk a rychlostí vyhledávání. V závěru je shrnutí výsledků, kterých bylo dosaženo při experimentech s kompaktním sufixovým automatem.

2 Analýza DNA

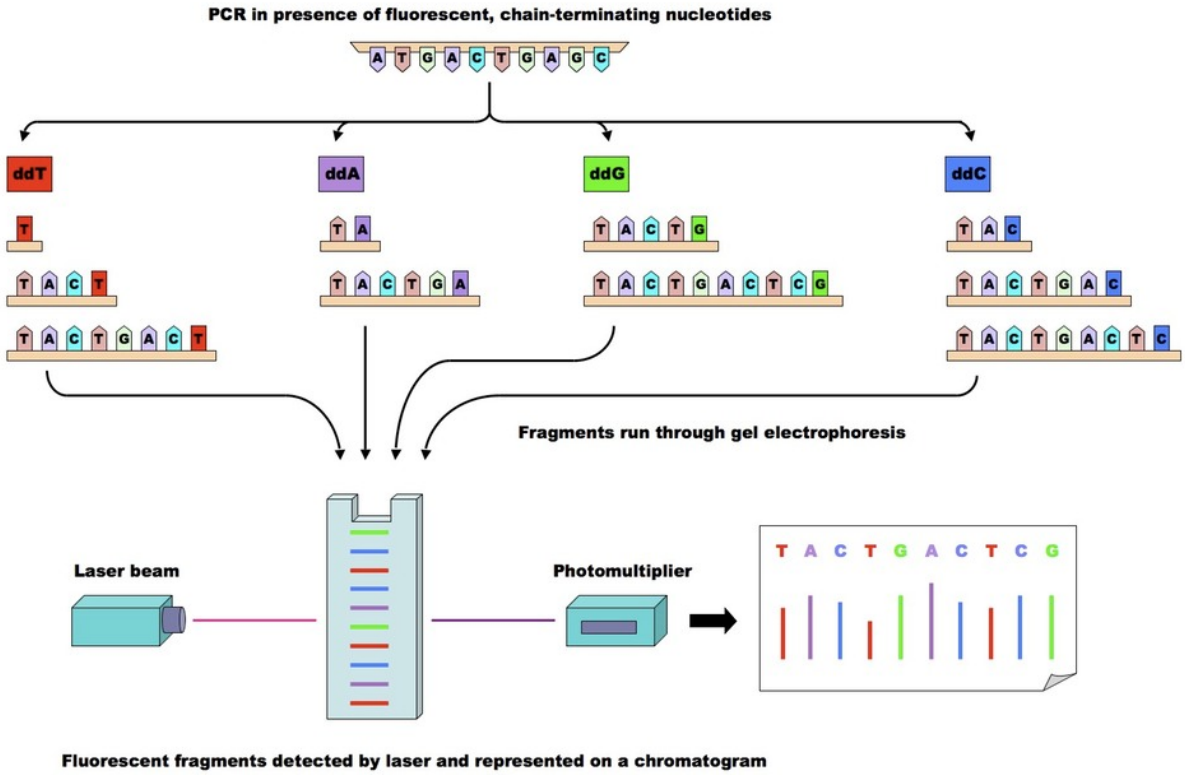
Genom je veškerá genetická informace organismu, obsahuje dědičné informace potřebné k jeho fungování. Chromozom je "balík", který tvoří část genomu. V jednoduchých formách života, jako jsou například bakterie, je celý genom zabalen do jediného chromozomu. Složitější organismy, jejichž genom je tisíckrát až milionkrát obsáhlejší než genom bakterií, mají genom rozdělený do několika různých chromozomů. Například u člověka to je 46 chromozomů, či 23 párů. Geny jsou základní jednotkou dědičnosti a nacházejí se v chromozomech. Různé geny určují různé charakteristiky či vlastnosti organismu. Bakterie obsahují několik stovek až tisíc genů, zatímco u člověka se odhaduje mezi 25 000 až 30 000 geny. DNA je molekula tvořící dědičný materiál v buňce, je to konkrétní nosič důležitých informací potřebných ke konstrukci organismu. Skládá se z menších jednotek nazývané nukleotidy ve 4 variacích - adenin, cytosin, guanin a thymin - často zkracované jako A, C, G, T. Geny jsou tvořeny z DNA a tím i celý genom je tvořen z DNA. První popis struktury dvoušroubovice DNA vznikl v roce 1953 biology *Watson a Crick*. Geny v lidském těle tvoří zhruba 25% lidské DNA. Studování DNA je klíčové pro pochopení nejen funkce jednotlivých genů, ale i ostatních částí genomu, pro které zatím není znám jejich význam [7].

2.1 Sekvenování genomu

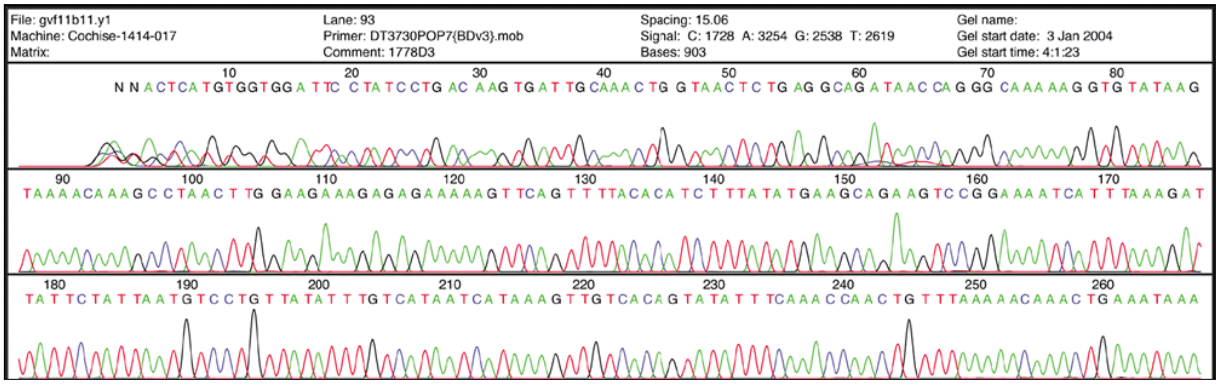
Od popsání struktury dvoušroubovice molekuly DNA uběhlo necelých 20 let než vznikly metody, které umožnily rozvoj výzkumu DNA. Těmito metodami jsou Maxam-Gilbert sekvenování [18] a Sanger sekvenování [21]. Sekvenování je proces určení přesného pořadí nukleotidů v dané molekule DNA. Využívá se k nalezení přesných sekvencí jednotlivých genů. Sangerova metoda se stala standardem ve výzkumné i komerční sféře právě kvůli technické nenáročnosti a spolehlivosti výsledků. Tato metoda se využívá dodnes právě pro sekvenování krátkých sekvencí a potvrzení výsledků získaných odlišnými metodami. Jedním z moderních přístupů k této metodě je využití fluorescenčních barviv, která se navážou na jednotlivé nukleotidy. Sekvence jsou poté posunovány skrze laserový paprsek, na jehož konci je detektor, který zpracovává změny ve vlnové délce laserového paprsku. Na obrázku 1 je zobrazeno schéma sekvenování touto metodou. Tento proces se nazývá elektroforéza, jehož výstupem detektoru je signál (elektroferogram), který dále zpracovává sekvenátor. V signálu se tvoří lokální maxima pro různé vlnové délky, která lze přiřadit jednotlivým nukleotidům. Na obrázku 2 je zobrazeno zpracování takového signálu.

Efektivně lze touto metodou sekvenovat části dlouhé několik stovek bází. Sekvence genů typicky obsahují několik tisíc bází. Nejdelší známý gen je spojen s Duchennovou muskulární dystrofií, který je dlouhý zhruba 2,4 miliónů bází. Pro zkoumání takto velikých genů bylo potřeba nalézt efektivnější metody.

Modifikováním Sangerovy metody vznikla metoda *shotgun sequencing* [22], volně přeloženo jako sekvenování nástřelem. Metoda je založena na náhodném rozdělení vstupní sekvence na krátké segmenty, které se poté naklonují a paralelně sekvenují pomocí Sangerovy metody. Tento



Obrázek 1: Schéma sekvenování DNA pomocí Sangerovy metody. Převzato z [2].



Obrázek 2: Vizualizace elektorferogramu při sekvenování DNA Sangerovou metodou. Převzato z [11].

postup je nutné několikrát opakovat, obecně se uvádí 20 až 30 krát, pro vygenerování dostatečně velkého počtu vzorků tak, aby vzniklo dostatečné pokrytí.

Takto získané fragmenty, které byly individuálně sekvenovány, je dále nutné zpětně spojit do původní sekvence. V těchto fragmentech se pak hledá cesta mezi překrytím fragmentů - *tiling path*. Z takto spojených fragmentů vzniknou kontigy, které mohou představovat část nebo i celek nějakého genu. Proces zarovnání a spojování takových fragmentů k získání původní sekvence se nazývá skládání sekvencí - *sequence assembly*. Data zpracovává software pro vyhledávání úseků řetězců, které se překrývají. Právě zde je důležité efektivně vyhledávat v textových řetězcích. Situaci značně komplikuje fakt, že se v genetických sekvencích velice často vyskytují opakované řetězce či genetické mutace. Právě kvůli mutacím zde nachází využití algoritmy pro přibližné vyhledávání v textu, které dovolují s určitou mírou nepřesnosti vyhledávat v řetězcích.

Dalším způsobem jak rozdělit genom a poskládat jej zpátky je metoda klonování - *clone-by-clone*. Principem této metody je rozdělení sekvence na dlouhé kusy, které se nazývají klony, dlouhé zhruba 150 000 párů bází. Pomocí genového mapování se pak zjišťuje, kde se daný klon nachází v genomu. Pak se každý klon rozdělí na menší části, které se překrývají, na délku zhruba 500 párů bází. Ve finále se tyto části sekvenují a protože se překrývají, lze sekvenci původního klonu zrekonstruovat.

2.2 Genové mapování

Genová mapa slouží k orientaci v genomu. Tato mapa je jedno-dimenzionální, stejně jako molekuly DNA, ze kterých se genom skládá. Obsahuje orientační body, kterými mohou být konkrétní krátké sekvence nukleotidů. Dále to mohou být oblasti, které ovládají chování genů, či geny samotné. Tyto orientační body jsou typicky kombinace písmen a čísel, které mohou představovat geny, například D14S72, GATA-P7042 [7].

Genové mapy se oproti genové sekvenci liší tím, že genové sekvence obsahují přesné pořadí všech bází DNA v genomu. Zatímco genová mapa obsahuje několik orientačních bodů v genomu. Mapování a sekvenování mohou být zcela nezávislé procesy. Například je možné určit pozici v genu, tedy namapovat gen, bez sekvenování daného genu. Mapa genomu nemusí nic vypovídat o konkrétní sekvenci genomu a sekvence nemusí říct nic o mapě genomu. Nebo se může stát, že orientační bod v mapě genomu je konkrétní DNA sekvence. Příkladem může být sekvence:

$$GCCATTGACGTCCCCTTGAAAGGAGAGAGAGACTCCGCATGCG$$

Genová mapa by mohla vypadat například takto:

GCC————CCCC————CTCCG——GCG

V této mapě jsou následující orientační body: GCC, CCCC, CTCCG, GCG. Ve stejné sekvenci jsou tyto báze orientačními body, jinými slovy je sekvence genová mapa s více informacemi. Převzato z [7].

Pro lidský genom a organismy s obsáhlým genomem obecně je vytváření genové mapy rychlejší a levnější než sekvenování celého genomu. Je to dáno tím, že pro mapování je potřeba zpracovat a zařadit mnohem méně informací, než v případě sekvenování.

Proč tedy rovnou nepřejít ke kroku sekvenování genomu? Genová mapa může pomoci při sekvenování. Jestliže se sekvenuje genom pomocí metody *clone-by-clone*, je potřeba znát genovou mapu k určení pozic, kam daný klon patří v genomu. Čím je tato mapa detailnější a přesnější, tím jednodušší je poskládat části genomu dohromady. S využitím metody *shotgun sequencing* genová mapa není potřeba, lze ji však pořád využít k přiřazení zpětně zrekonstruovaných sekvencí na patřičnou pozici do genomu.

To však neznamená, že genová mapa ztrácí smysl. Genová sekvence sama o sobě moc neříká, je to pouze sekvence 3 miliard (v případě lidského genomu) nukleotidů, které nenesou další informace. Při pohledu na sekvenci nelze jen tak říct, kde se může nacházet a jakou funkci může vykonávat, které úseky tvoří geny a které úseky jsou naopak nezajímavé. Orientační body v genové mapě poskytují určitou nápovědu k nalezení důležitých úseků v genové sekvenci.

2.3 Projekt Lidského Genomu

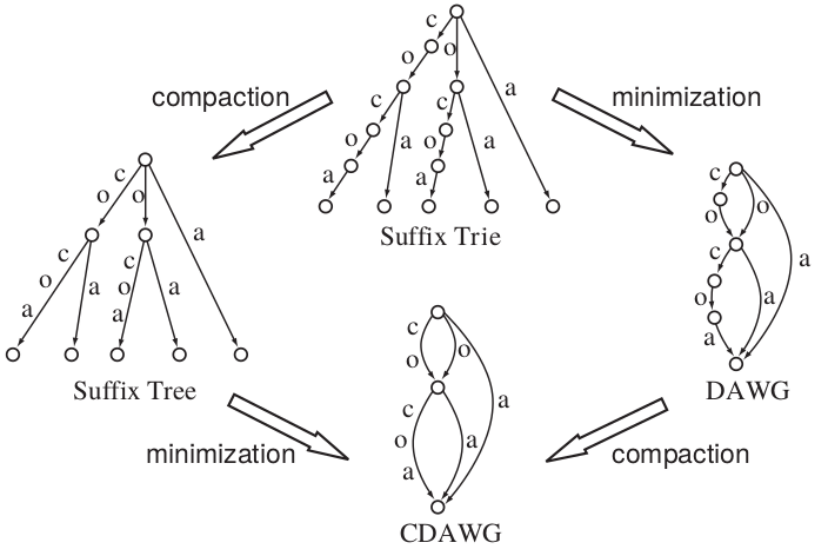
Projekt lidského genomu, anglicky Human Genome Project (HGP) byl mezinárodní výzkumný projekt započatý v roce 1990, ukončený v roce 2003, který byl financován americkou vládou. Do projektu se přihlásilo zhruba 100 dobrovolníků, kteří darovali vzorky DNA, z nichž bylo vybráno pouze pár vzorků. Cílem projektu bylo zmapovat lidský genom, nalézt původy genetických onemocnění a následný výzkum léčby těchto nemocí. Pro sekvenování lidského genomu byly použity jak metody *shotgun sequencing*, tak genové mapování. Ve výsledku byla vytvořena referenční sekvence, pomocí níž lze identifikovat variace v genomech, které zvyšují rizika častých onemocnění jako je diabetes nebo rakovina [10]. V současnosti existují další projekty, jako je například Human Genome Organization (HUGO).

3 Využívané datové struktury

V oblasti genetického sekvenování jsou často vyhledávány krátké řetězce, které je potřeba zarovnat na referenční sekvenci. Protože lidský genom se skládá ze zhruba 3 miliard znaků, je nezbytné provádět efektivní vyhledávání nad velmi dlouhými řetězci pro krátké sekvence. V této kapitole jsou popsány typické datové struktury, sufixové stromy a sufixová pole, které jsou používány nejen v bioinformatice, ale obecně ve vyhledávání řetězců a zpracování textu.

3.1 Sufixový strom

Nejprve je potřeba se zmínit o sufixové trii. Je to stromová struktura, pomocí níž lze reprezentovat všechny možné sufixy daného řetězce. Pokud máme vytvořený takovýto index nad vstupním řetězcem, můžeme efektivně provádět operace pro jeho podřetězce. Minimalizací sufixové trie lze získat sufixový strom, který v praktickém využití nahrazuje sufixové trie, protože lze sufixové stromy efektivněji reprezentovat v paměti. Typické operace nad sufixovou trií a sufixovým stromem jsou vyhledání řetězců, nalezení nejdelšího společného podřetězce dvou řetězců, či přibližné vyhledávání v textu (*fuzzy searching*)[25]. Na obrázku 3 je přehled indexových struktur založených na sufixové trii. O sufixových automatech $DAWG$ a $CDAWG$ pojednávají pozdější kapitoly 4.1 a 4.2.



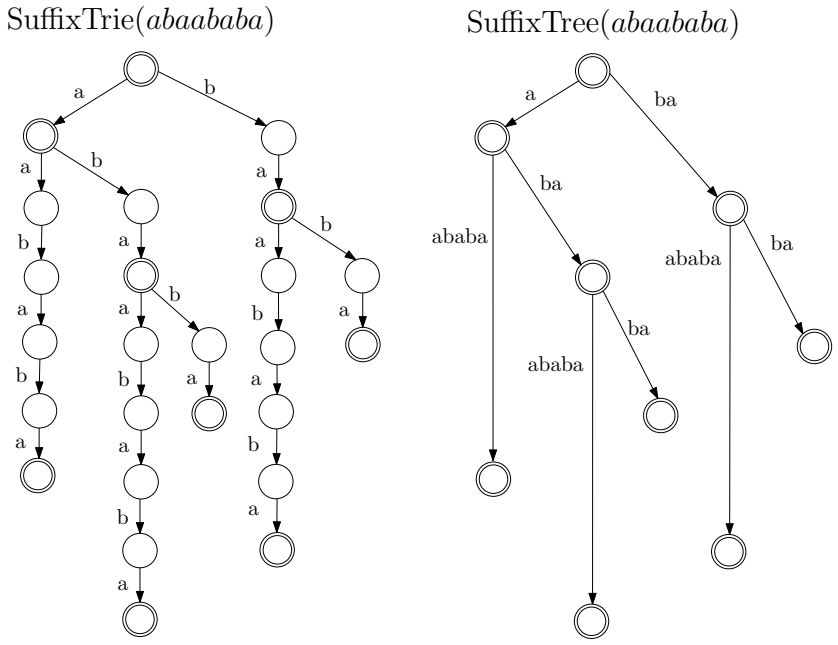
Obrázek 3: Převod mezi sufixovou trií $STrie$, sufixovým stromem ST, sufixovým automatem $DAWG$ a kompaktním sufixovým automatem $CDAWG$. Převzato z [13].

3.1.1 Definice

Nechť Σ je neprázdná abeceda a Σ^* je množina řetězců nad abecedou Σ, ε je řetězec nulové délky nad abecedou Σ a \$ je speciální znak, který se nevyskytuje v abecedě Σ. Dále nechť je

řetězec $S = s_1, s_2, \cdots s_n$, kde $S \in \Sigma^*$, $n = |S|$ je délka řetězce S, dále S_i je znak na pozici i a zápis $S_{i..j}$ je podřetězec $S_i, S_{i+1}, .., S_j$. Jestliže $S = xyz$, kde $x, y, z \in \Sigma^*$, potom x, y, z jsou podřetězci řetězce S, x je prefix řetězce S a z je sufix řetězce.

Sufixová trie $STrie(S)$ řetězce S je stromová struktura, která indexuje všechny sufixy řetězce S. V sufixové trii je právě jeden kořen stromu, hrany mezi vrcholy jsou popsány právě jedním znakem. Nalezení podřetězce X v řetězci S, či určení, zda X je sufixem S, lze pomocí indexových struktur založených na sufixové trii uskutečnit v čase lineárním s délkou hledaného řetězce, tedy $\mathcal{O}(|X|)$. Naivní přístup k tomuto problému, tedy porovnávat řetězce X s S znak po znaku, je v průměru v čase $\mathcal{O}(|X| + |S|)$, v nejhorším případě $\Omega(|X| * |S|)$. Výhody těchto indexových struktur jsou patrné při velmi dlouhém vstupním řetězci S, ve kterém se vyhledává.



Obrázek 4: Sufixová trie $STrie(abaababa)$ a sufixový strom $ST(abaababa)$. Vrcholy v sufixové trii, které jsou zobrazené dvojitou čarou, jsou potřebné vrcholy, které nejsou sloučeny v sufixovém stromu. Zároveň určují vrcholy, které představují sufix řetězce *abaababa*.

Sufixovou trii lze minimalizovat, to znamená, že posloupnosti vrcholů, které mají právě jednu výchozí hranu, jsou sloučeny. Hrany jsou popsány řetězcem, který odpovídá zřetězením znaků získaných při sloučení těchto vrcholů. Vznikne kompaktní sufixová trie, tedy sufixový strom $ST(S)$. Na obrázku 4 je zobrazena sufixová trie $STrie(abaabab)$ a sufixový strom $ST(abaabab)$, který lze získat z takové trie. U sufixových stromů a trií jsou tradičně cesty doplněny o speciální znak určující konec řetězce, například \$. V ilustracích v této práci je aplikován koncept přijímacích stavů z teorie automatů při vizualizaci stromových struktur. U vrcholů, které jsou zobrazeny dvojitou čarou, si lze domyslet prázdnou hranu, která určuje konec řetězce \$. Je to právě kvůli

podobnosti těchto struktur na sufixové automaty, které budou popsány později v této práci.

Sufixový strom řetězce S, zapsáno jako $ST(S)$, je kompaktní trie všech sufixů řetězce S, kde:

- Každý vnitřní vrchol stromu má alespoň dva potomky (kromě kořene).
- Každá hrana je označena podřetězcem e nenulové délky řetězce S, kde $|e| \geq 1$. Je to právě díky kompaktní vlastnosti stromu, kde hrany nemusí být popsány právě jedním znakem, ale řetězcem.
- Žádné dvě sousedící hrany pro daný vrchol nejsou označeny stejným řetězcem.
- Každý list stromu je označen pozicí v řetězci S, kde list je označení pro vrchol stromu, který nemá další potomky.
- Řetězec získaný průchodem grafu z kořene stromu do vrcholu stromu je podřetězec řetězce S.

Sufixový strom obsahuje právě $2n$ vrcholů, kde n je délka řetězce S, počet listů je n, kde každý list odpovídá právě jednomu sufixu řetězce S, počet vrcholů je $n - 1$ a právě jeden kořen stromu.

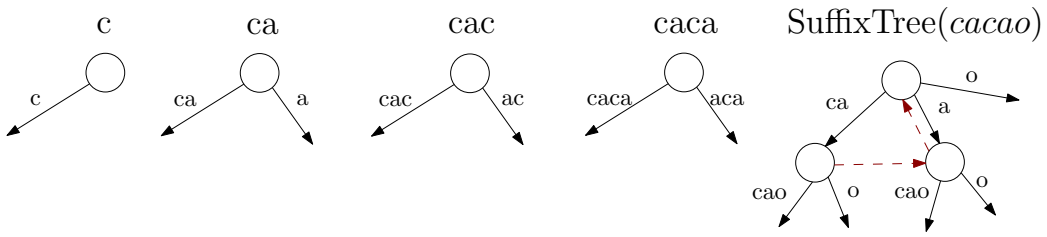
Pojem $suf(v)$ je definován jako *sufixový odkaz vrcholu v*. Každý vrchol ve stromu má sufixový odkaz na nějaký další vrchol, avšak v některé literatuře se nedefinuje sufixový odkaz pro kořen stromu. Sufixové odkazy jsou využívány v algoritmech konstrukce sufixových stromů, kde slouží pro efektivní přesun mezi izomorfními podstromy[5]. Jsou klíčové k lineární časové složitosti algoritmu konstrukce.

Při vizualizaci sufixových stromů se hrany popisují řetězci e pro ilustraci. V praxi se tyto hrany nepopisují samotnými řetězci, nýbrž indexem a a hodnotou b. Index a ukazuje na počáteční pozici řetězce e v S, tedy S_a; hodnota b je délka e.

3.1.2 Konstrukce sufixového stromu

Algoritmy konstrukce sufixového stromu lze rozdělit do tří hlavních kategorií [17]:

V paměti (In-memory): Vstupní řetězec S a výsledný sufixový strom se vejdou do paměti zařízení. Do této kategorie spadá algoritmus Ukkonen [26], který pracuje v čase $\mathcal{O}(n)$ pro konstantní abecedu a $\mathcal{O}(n \log n)$ pro obecnou abecedu. Výhodou algoritmu je, že nepotřebuje na počátku znát celý svůj vstup S, proto je označen za *online* algoritmus. Jakmile se však sufixový strom nevleze do hlavní paměti, algoritmy v této kategorii provádějí v průměru $\mathcal{O}(n)$ přístupů na disk, které jsou pomalé. Algoritmus McCreight [19] pracuje rovněž v lineárním čase, dokáže však lépe pracovat s pamětí. Postupně přidává do stromu všechny sufixy řetězce, od nejdelšího po nejkratší. Potřebuje znát na svém vstupu celý vstupní řetězec S, označuje se za *offline* algoritmus. Oba tyto algoritmy využívají konceptu sufixových odkazů při konstrukci stromu, které umožňují algoritmům pracovat v lineárním čase. Na obrázku 5 je příklad konstrukce $ST(cocoa)$ podle algoritmu Ukkonen.



Obrázek 5: Kroky konstrukce sufixového stromu $ST(cacao)$ podle algoritmu přímé konstrukce Ukkonen. Každý krok představuje zpracování dalšího znaku.

V algoritmu Ukkonen jsou hrany, které vedou do konce zpracovaného řetězce, označeny jako otevřené. Nemusí tedy explicitně vézt do nějakého vrcholu, je tím chápán implicitní vrchol, který představuje zbytek řetězce. V obrázku je tato myšlenka zachována vzhledem k algoritmu. Detaily k algoritmu Ukkonen a jeho implementaci lze nalézt v [26].

"Semi-disk-based": Tyto algoritmy rozdělí výsledný sufixový strom na několik menších podstromů, které jsou zapsány na disk. Do teto kategorie patří algoritmy Hunt [12], TDD [23], ST-Merge [24], TRELLIS [20]. Algoritmus TRELLIS funguje na principu rozdělení vstupního řetězce na několik podřetězců, pro každý takový podřetězec se nezávisle sestaví sufixový strom a zapíše se na disk. V druhé fázi se tyto podstromy sloučí do finálního sufixového stromu. Přestože je časová složitost algoritmu $\mathcal{O}(n^2)$, ve výsledku je tento algoritmus rychlejší než Ukkonen právě díky nízké četnosti přístupu na disk.

"Out-of-core": Algoritmy v této kategorii podporují řetězce, které se nevejdou do hlavní paměti, avšak snaží se minimalizovat náhodné přístupy na disk. Nejznámějšími jsou algoritmy B^{2}ST [3], který je založen na sufixových polích a algoritmus Wavefront [8].

Jsou rovněž známy algoritmy pro paralelní konstrukci sufixových stromů [9, 1, 17].

3.2 Sufixové pole

Sufixové pole je datová struktura, která reprezentuje všechny sufixy daného řetězce, které jsou lexikograficky uspořádané. Výhodou sufixového pole je až 5x nižší paměťová náročnost oproti sufixovému stromu (záleží na implementaci sufixového stromu).

3.2.1 Definice

Využijeme definici abecedy Σ, řetězce S, podřetězce X z kapitoly sufixových stromů. Sufixové pole SA řetězce S je pole nezáporných celých čísel, která odpovídají počátečním pozicím všech lexikograficky seřazených sufixů řetězce S. Hodnota v sufixovém poli na pozici i, $SA[i]$, je počáteční pozice sufixu i-tého nejmenšího sufixu řetězce S.

Sufixové pole lze v paměti reprezentovat jediným souvislým polem hodnot. Daný řetězec S obsahuje právě $|S|$ sufixů, sufixové pole pro řetězec S bude obsahovat $n = |S|$ prvků a potřebuje

n paměti. Oproti tomu sufixový strom potřebuje pro reprezentaci jednoho sufixu až $5n$ paměti. Vyhledávání v sufixovém poli využívá faktu, že sufixy jsou lexikograficky seřazeny s rostoucí délkou sufixu. Pomocí dvou binárních vyhledávání, každé v čase $\mathcal{O}(m \log n)$, kde $m = |X|$ a $n = |S|$, získáme první a poslední výskyty řetězce. V tabulce 1 je ukázkové sufixové pole pro řetězec $SA(banana)$.

Například pro nalezení všech výskytů řetězce $X = na$ v $S = banana$: binárním vyhledáváním se zjistí indexy L, R, které jsou v rozmezí $0 \ldots |S|$. Konkrétně pro $X = na$ bude $L = 5$ a $R = 6$. Počáteční indexy všech výskytů řetězce na se nachází v poli A na indexech $A[L], A[L + 1], \ldots A[R]$; pro konkrétní příklad to jsou indexy $5, 6$. Řetězec X se tedy nachází na pozicích $A[5] = 4$ a $A[6] = 2$.

Tabulka 1: Sufixové pole pro $S = banana$

i	0	1	2	3	4	5	6
$S[i]$	b	a	n	a	n	a	\$
$A[i]$	6	5	3	1	0	4	2
0	\$	a	a	a	b	n	n
1		\$	n	n	a	a	a
2			a	a	n	\$	n
3			\$	n	a		a
4				a	n		\$
5				\$	a		
6					\$		

3.2.2 Konstrukce

Naivní metoda konstrukce sufixového pole je nejprve zkonstruovat pole všech sufixů a následně toto pole lexikograficky uspořádat. Časová složitost je $\mathcal{O}(n^2 \log n)$, pokud předpokládáme, že algoritmus řazení pracuje v čase $\mathcal{O}(n \log n)$ a každé porovnání dvou řetězců znamená $\mathcal{O}(n)$.

Dalším způsobem konstrukce je nejprve zkonstruovat sufixový strom pomocí některého z algoritmů popsaných v kapitole sufixových stromů 3.1.2. Sufixové pole lze ze sufixového stromu získat průchodem grafu do hloubky, kde jednotlivé hrany jsou procházeny v lexikografickém uspořádání.

Manber & Myers vytvořili přímý algoritmus konstrukce v čase $\mathcal{O}(n \log n)$ [16]. Později byl překonán algoritmem Skew, který pracuje v čase $\mathcal{O}(n)$ [15]. V praxi je potřeba k sufixovému poli znát i pole nejdelších společných prefixů (LCP - Lowest Common Prefix). Jsou známy algoritmy pro lineární konstrukci LCP pole [15, 14].

4 Návrh a implementace sufixových automatů

Sufixové automaty $DAWG$ jsou datové struktury založené na sufixové trii a sufixovém stromu. Přináší úspornější využití paměti v porovnání se sufixovým stromem. Tato kapitola nejprve popíše princip sufixových automatů a rozdíly oproti sufixovým stromům. Poté se zaměří na kompaktní sufixové automaty, na které je tato práce zaměřena. Dále je popsána konkrétní implementace kompaktních sufixových automatů, způsoby zápisu a komprese dat na disk. Následuje popis vyhledávání řetězců v automatu a diskuze o možnostech paralelního zpracování v konkrétní implementaci.

4.1 Sufixový automat

Sufixový automat $DAWG$ (*Directed Acyclic Word Graph*) je acyklický deterministický automat, jehož přijímací stavy představují sufixy řetězce S. Lze jej zkonstruovat nejprve konstrukcí sufixové trie a následným spojením izomorfních podstromů. Takto spojené podstromy dávají sufixovému automatu vlastnost, že potřebuje méně paměti vzhledem k stromům. Tento proces se nazývá minimalizace a odpovídá minimalizaci konečného deterministického automatu.

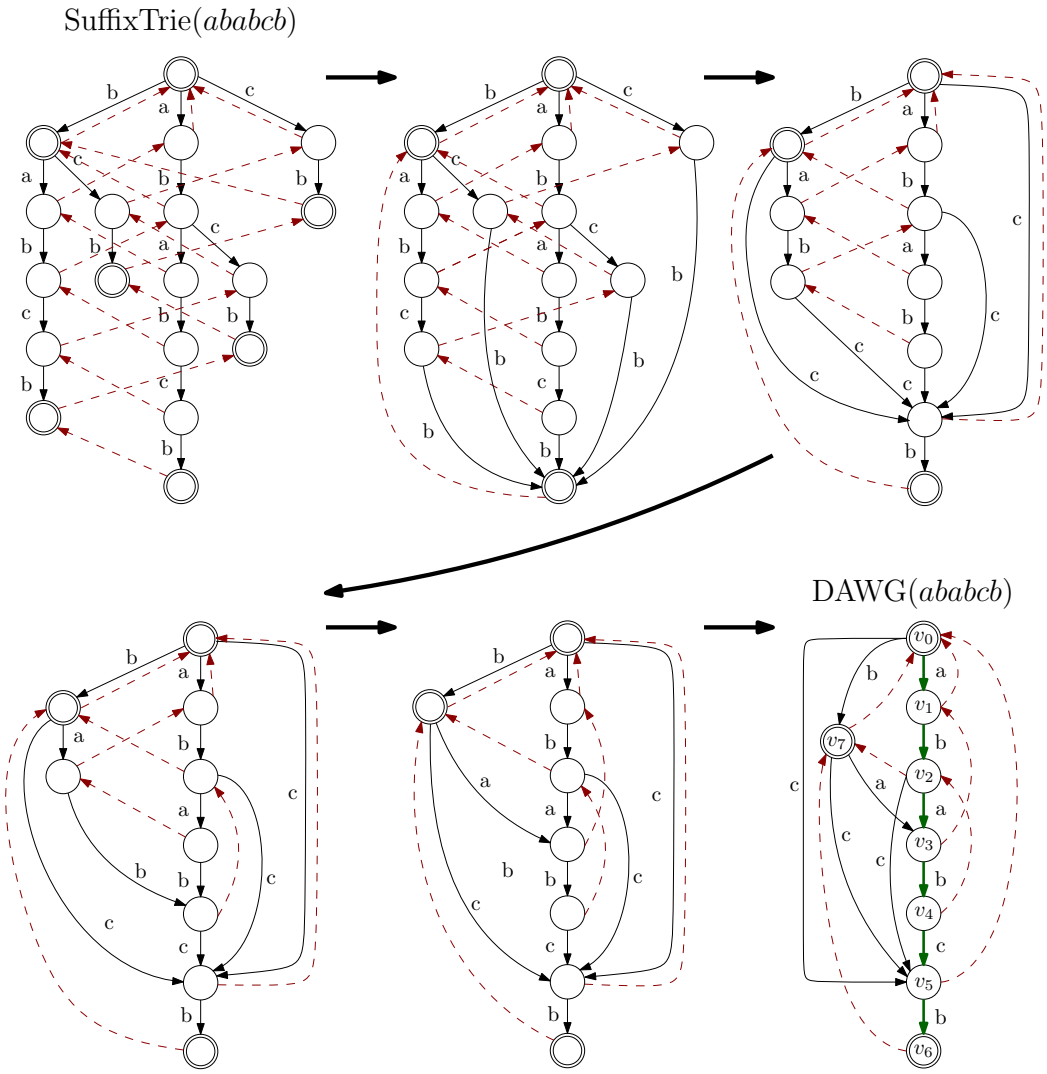
Na obrázku 6 je příklad sufixového automatu $DAWG(ababcb)$, spolu s jednotlivými kroky při převodu sufixové trie na sufixový automat; $DAWG$ lze získat ze sufixové trie v čase $\mathcal{O}(|S|)$. Pokud bychom při minimalizaci sufixové trie ignorovali přijímající vrcholy trie, získali bychom minimální automat, který sice přijímá všechny podřetězce x řetězce S, ale nemusí přijímat všechny sufixy[5]. Obrázek 7 zobrazuje takový minimálního automat, který není platným $DAWG$.

4.1.1 Definice

Využijeme definici abecedy Σ, řetězce S, podřetězce X z kapitoly sufixových stromů. Dále $Suffix(S)$ je množina všech sufixů řetězce S a $F(S)$ množina všech podřetězců řetězce S. Pro daný automat je n-tice (p, a, q) přechod ze stavu p přes hranu s popisem a do stavu q. V případě, kdy je hrana popsána jedním znakem, je použito latinské písmeno (např. a); pro přechody řetězci delší než jeden znak je použit znak řecké abecedy (např. α). Zápis $(p, \alpha]$ znamená přechod ze stavu p přes hranu, kde α je prefix řetězce, který takovou hranu popisuje. Následující definice $DAWG(S)$ je převzata z [5].

Sufixový automat řetězce S, zapsáno $DAWG(S)$, je deterministický automat, který přijímá $Suffix(S)$. Příklad $DAWG$ pro řetězec $S = ababcb$ je na obrázku 6. Stavy, které jsou zobrazeny dvojitou čarou, jsou přijímající. Takové stavy přijímají sufix, který lze získat průchodem automatu přes jeho hrany. Pokud nás zajímá, zda se řetězec x nachází v S, lze pohlížet na všechny stavy jako přijímající, musí však v automatu existovat cesta, která odpovídá řetězci x. V literatuře se vstupní stav často označuje pojmem *root*, výstupní stav $DAWG$ pojmem *sink*.

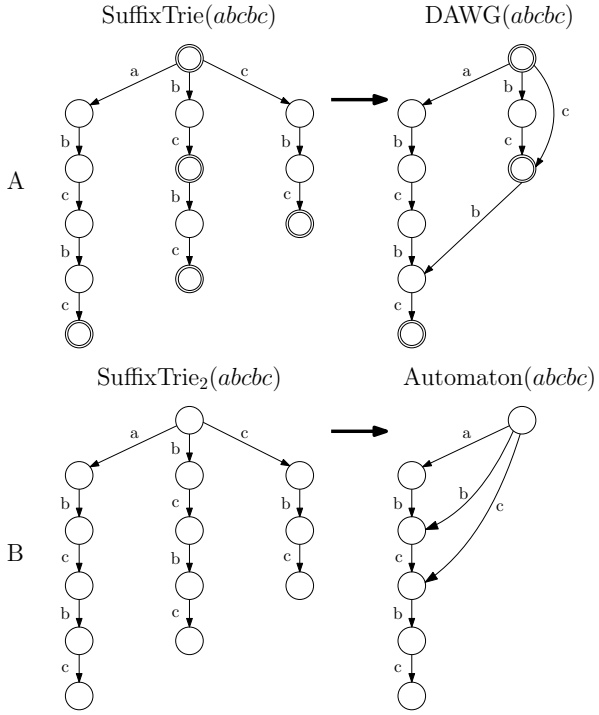
Velikost $DAWG(S)$ je $\mathcal{O}(|S|)$ a automat lze zkonstruovat v čase $\mathcal{O}(|S|)$. Maximální počet stavů takového automatu je $2|S| - 1$ a maximální počet hran je $3|S| - 4$.



Obrázek 6: Konstrukce sufixového automatu $DAWG(ababcb)$ spojením izomorfních podstromů sufixové trie $ST(ababcb)$.

Mějme podřetězec x řetězce S. Funkce $endpos(x)$ vrací množinu všech pozic v řetězce S, kde výskyt x končí. Dále y je další takový podřetězec řetězce S. Podstromy sufixové trie $Trie(S)$ s kořeny na pozicích x a y jsou izomorfní právě tehdy, když $endpos(x) = endpos(y)$. V $DAWG(S)$ jsou cesty v grafu, pro které platí tato rovnost, zakončeny ve stejném stavu. Stavy v $DAWG$ odpovídají neprázdným množinám konečných pozic $endpos(x)$. Kořen $DAWG$, tedy vstupní stav I, odpovídá celé množině pozic $0, 1, 2, ..., N$. Na tyto množiny je pohlíženo pouze teoreticky při konstrukci, v praxi tyto množiny nejsou nikde uchovávány.

Každému stavu v v $DAWG$ je přiřazena hodnota $val(v)$, která je rovna nejdelšímu řetězci, který lze získat průchodem $DAWG$ do stavu v. Všechny stavy v $DAWG$ jsou třídami ekvivalence listů v sufixové trii, kde ekvivalence znamená izomorfismus podstromů v trii. V třídě ekvivalence

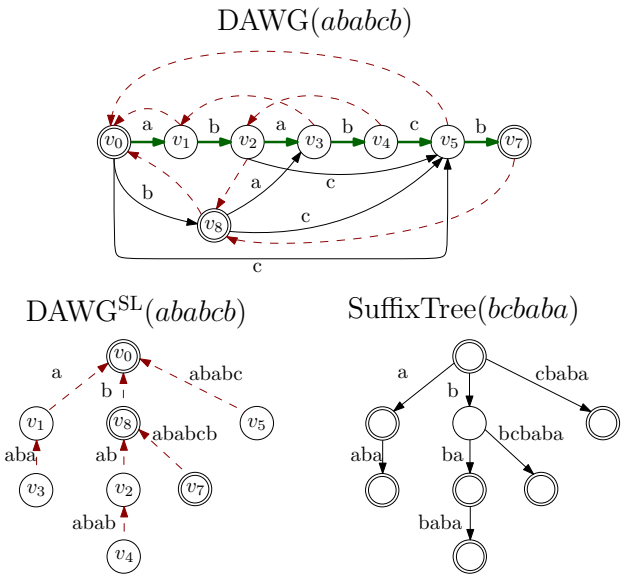


Obrázek 7: V části A je zobrazen korektní převod sufixové trie $STrie(abcbc)$ do $DAWG$. V části B jsou přijímající vrcholy trie ignorovány při minimalizaci sufixové trie $SuffixTrie_2$. Vznikne minimální automat $Automaton(abcbc)$, který přijímá všechny podřetězce, ale není platný sufixový automat. Převzato z [5].

pro stav v je $val(v)$ její nejdelší reprezentant.

Nechť v je stav v $DAWG(S)$ odlišný od vstupního stavu. Definujeme stav $w = suf(v)$, kde $val(w)$ je nejdelší sufix $val(v)$, který odpovídá jinému stavu než v. Protože podřetězec pro $suf(v)$ je vždy kratší než $suf(v)$, sufixové odkazy indukují stromovou strukturu na stavech $DAWG$. Stav $suf(v)$ je brán jako předek stavu v v tomto stromu. Sufixové odkazy jsou zobrazeny například v obrázku 6 přerušovanou červenou čarou. Cesta v $DAWG$, která vede ze stavu, který přijímá celý vstup *ababcb*, do vstupního stavu přes sufixové odkazy, určuje přijímající stavy. Podobnou vlastnost lze vypozorovat i u sufixové trie, sufixového stromu či kompaktního sufixového automatu.

Sufixový strom a sufixový automat mají podobné vlastnosti, protože jsou obě tyto struktury kompaktní reprezentace stejné sufixové trie. Vezměme $DAWG(ababcb)$ a uspořádejme stavy podle sufixových odkazů. Tyto hrany popišme nejdelšími řetězci, které lze získat průchodem automatu do daného stavu. Získáme stromovou strukturu, která v podstatě odpovídá sufixovému stromu pro inverzní řetězec S^R, tedy $ST(bcbaba)$, viz. obrázek 8.



Obrázek 8: Strom sufixových odkazů pro $DAWG(ababcb)$ je sufixový storm $ST(ababcb^R)$. Strom sufixových odkazů pro $DAWG$ má přechody popsány nejdelším řetězcem, který lze získat průchodem $DAWG$ do daného cílového stavu hrany. U sufixového stromu jsou hrany popsány řetězci získanými průchodem stromu.

4.1.2 Konstrukce

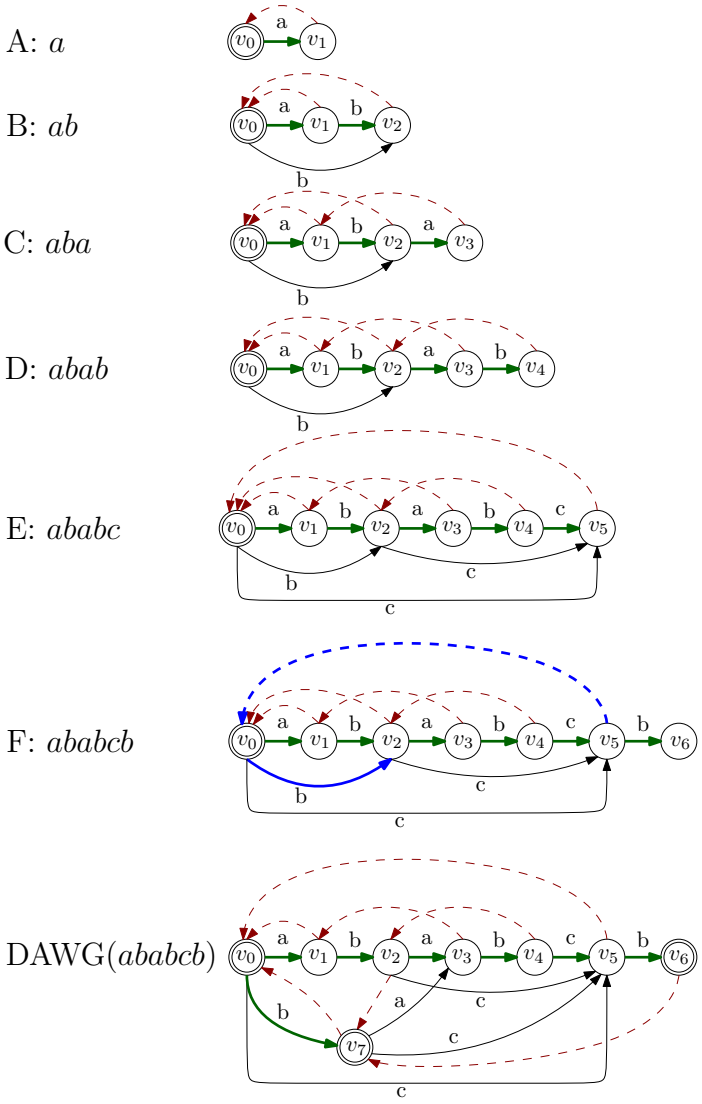
Algoritmus přímé konstrukce pracuje $on-line$, to znamená, že zpracovává vstupní řetězec zleva doprava a na počátku algoritmu nemusí znát celý vstup. Pseudokód tohoto algoritmu je popsán ve výpise 1. Při konstrukci se rozlišuje mezi dvěma typy hran (přechody mezi stavy): pevné hrany (v literatuře *solid edges*) a obyčejné hrany. V ilustracích jsou pevné hrany vyobrazeny tučnou zelenou čarou. Hrany jsou označeny jako pevné, pokud jsou součástí nejdelší cesty v automatu z kořene. Jinými slovy slovy lze říct, že hrany, které nejsou pevné, představují v automatu zkratky mezi stavy. Nazývají se pevné, protože jakmile jsou vytvořeny, v průběhu konstrukce $DAWG$ se tyto hrany nemění. Naopak hrany, které nejsou pevné, se mohou v dalších krocích změnit.

Jednotlivé kroky pro konstrukci $DAWG(ababcb)$ jsou zobrazeny v obrázku 9. Důležitý je krok F_1 - přidáním znaku b za $DAWG(ababc)$. V tomto kroku je cesta algoritmu znázorněna tučnou modrou čarou. Stavy, které odpovídají sufixům řetězce $ababc$, tedy v_0 a v_5, nyní potřebují přidat hranu b. Stav v_0 má hranu b, která není pevná, tvoří zkratku mezi stavy v_0 a v_2, která obchází stav v_1 a hrany ab. Oba podřetězce b a ab vedou do stavu v_2 jako přijímající, ale ab není sufixem $ababcb$. Proto je stav v_2 rozdělen na stav v_7, sufixový odkaz z v_2 je přesměrován na v_7, stejně jako sufixový odkaz z nového stavu v_6.

V obrázku 10 je příklad, jak by automat vypadal, kdyby se neprovedl krok F, kde se stav v_2 rozdělí. Za $DAWG(ababcb)$ je přidán další znak e. Vznikne tak cesta abe, na obrázku zobrazena

```
1: procedure ConstructCDAWG
2:     root ← sink
3:     suf(root) = null
4:     for i ← 1 To n do
5:         a = text[i]
6:         create new node newsink
7:         make a solid a-edge (sink, newsink)
8:         w ← suf(sink)
9:         while (w ≠ null) And (son(w, a) = null) do
10:            make a non-slid a-edge (w, newsink)
11:            w ← suf(w)
12:        end while
13:        v ← son(w, a)
14:        if w = null then
15:            suf(newsink) ← root
16:        else if (w, v) is a solid edge then
17:            suf(newsink) ← v
18:        else
19:            // split the node v
20:            create a node newnode, copy outgoing edges from v, make them non-solid
21:            change (w, v) into a solid edge (w, newnode)
22:            suf(newsink) ← newnode
23:            suf(newnode) ← suf(v)
24:            suf(v) ← newnode
25:            w ← suf(w)
26:            while w ≠ null And (w, v) is a non-solid a-edge do
27:                redirect this edge to newnode
28:                w ← suf(w)
29:            end while
30:        end if
31:        sink ← newsink
32:    end for
33: end procedure
```

Výpis 1: Algoritmus přímé konstrukce $DAWG$.

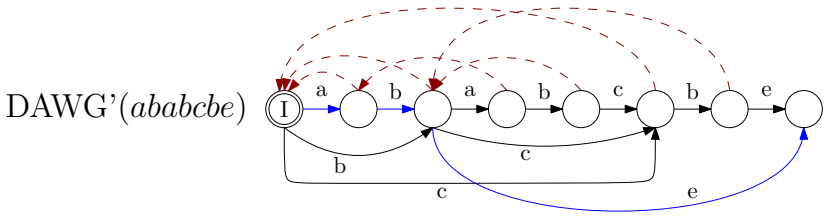


Obrázek 9: Jednotlivé kroky přímé konstrukce $DAWG(ababcb)$

modrými čarami, která není platným sufixem řetězce *ababcbe*. Automat v tomto případě není platným sufixovým automatem $DAWG$.

4.2 Kompaktní sufixový automat

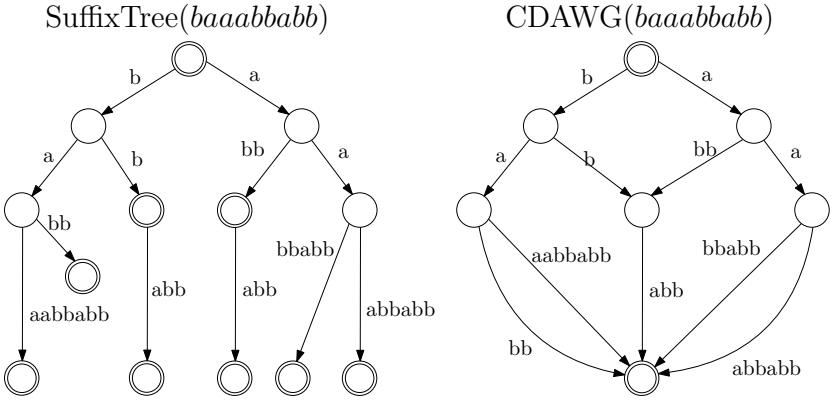
Kompaktní sufixový automat $CDAWG(S)$ (Compact Directed Acyclic Word Graph) je sufixový automat, který vznikne kompresí sufixového automatu $DAWG$. Stejně jako $DAWG$ přijímá sufixy $Suf(X)$ řetězce S. Podle experimentů se sufixovými automaty na DNA sekvencích je zhruba 60% stavů, ze kterých vede právě jedna hrana. Komprese $DAWG$ do $CDAWG$ proto přinese důležitou úsporu místa [4]. $CDAWG$ lze rovněž získat minimalizací sufixového stromu



Obrázek 10: Automat, který není korektní $DAWG$. Vznikne v případě, že se vynechá rozdělení pro přechod b v kroku F v obrázku 9. Modře je zvýrazněna cesta v automatu pro řetězec abe, který není platným sufixem řetězce $ababcbe$.

spojením všech izomorfních podstromů, viz obrázek 11. Pro představu je na obrázku 3 přehled převodů mezi těmito strukturami.

Blumer *et al* [4] prezentovali algoritmus konstrukce $CDAWG$, který nejprve musí zkonstruovat $DAWG$, po jehož kompresi vznikne $CDAWG$. Algoritmus komprese je založen na průchodu automatu $DAWG$ do hloubky a pracuje v lineárním čase s počtem stavů v $DAWG$. Výsledkem této komprese je kompaktní sufixový automat $CDAWG$, ve kterém jsou hrany popsány řetězci, zatímco v $DAWG$ jsou hrany popsány právě jedním znakem. Zároveň jsou z $DAWG$ odebrány stavy, které nejsou přijímající a ze kterých vede právě jedna hrana; vstupní stav a *sink* stav jsou přijímající, nejsou proto odebrány. Algoritmus této komprese je popsán ve výpise 2. Protože lze $DAWG$ zkonstruovat v lineárním čase podle délky vstupního řetězce S, je i tento algoritmus v čase $\mathcal{O}(|S|)$. Tento algoritmus konstrukce $CDAWG$ závisí na $DAWG$, který v praxi potřebuje zhruba dvojnásobek paměti [4, 6]. Je proto efektivnější se zaměřit na přímou konstrukci $CDAWG$.

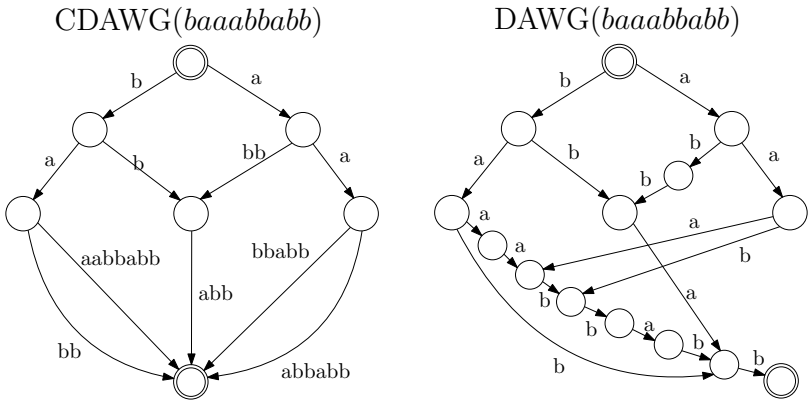


Obrázek 11: Minimalizace sufixového stromu $ST(baaabbabb)$ do $CDAWG(baaabbabb)$. Převzato z [5].

```
1: procedure REDUCTION(state E) returns (ending state, length of redirected edge)
2:     if E not marked then
3:         for all existing edge (E, a] do
4:             (state(E, a], |label((E, a])|) ← REDUCTION(state(E, a])
5:         end for
6:         mark(E) ← TRUE
7:     end if
8:     if E is outdegree one then
9:         Let (E, a] ← this edge
10:        return (state[E, a], +|label((E, a])|)
11:    else
12:        return (E, 1)
13:    end if
14: end procedure
15: REDUCTION(root_state)
```

Výpis 2: Algoritmus konstrukce $CDAWG$ kompresí $DAWG$.

Jelikož je znám algoritmus převodu $DAWG$ na $CDAWG$, lze i zpětně převézt $CDAWG$ na $DAWG$. Pro každý vrchol v $CDAWG$ je vybrána nejdelší hrana. Tato hrana se rozdělí, pro každý znak v hraně vznikne nový stav, a všechny ostatní hrany vedoucí do původního stavu jsou přesměrovány. Na obrázku 12 je zpětný převod $CDAWG(baaabbabb)$ do $DAWG(baaabbabb)$ tímto způsobem.



Obrázek 12: Dekompresí $CDAWG(baaabbabb)$ lze zpětně získat $DAWG(baaabbabb)$.

4.2.1 Přímá konstrukce $CDAWG$

Algoritmus přímé konstrukce $CDAWG$ uvedli v roce 1997 Crochemore a Vérin [6]. Protože lze $CDAWG$ zkonstruovat minimalizací sufixového stromu, je tento algoritmus založen na al-

goritmu konstrukce sufixového stromu McCreight [19]. Ve výpise 3 je algoritmus přímé konstrukce $CDAWG$, který pracuje v lineárním čase $\mathcal{O}(|S|)$. V podstatě algoritmus přímé konstrukce $CDAWG$ vkládá cesty odpovídající všem sufixům S od nejdelšího po nejkratší. Pro vysvětlení konkrétních kroků v algoritmu lze odkázat na [6].

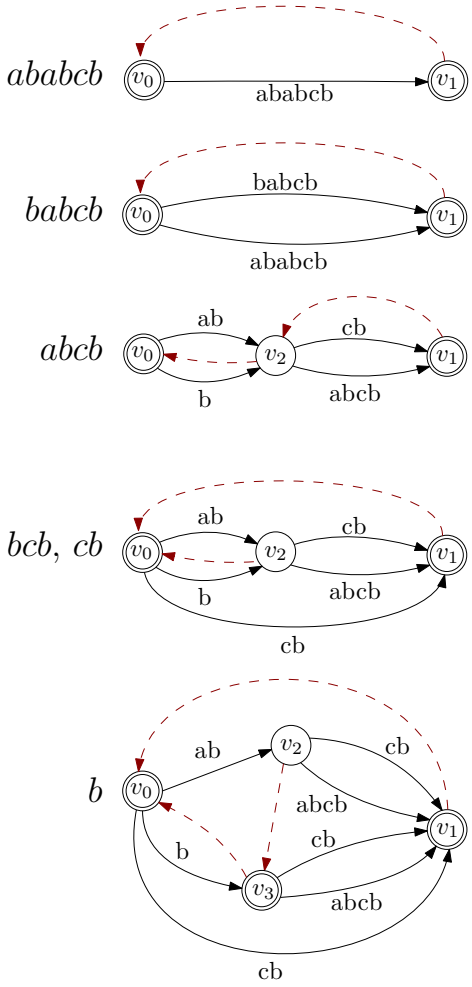
```
1: p ← I
2: i ← 0
3: while not end of x do
4:     (q, γ) ← SlowFind(p)
5:     if γ = ϵ then
6:         INSERT(q, tail, F)
7:         s_x(F) ← q
8:         if q ≠ I then
9:             p ← s_x(q)
10:        else
11:            p ← I
12:        end if
13:    else
14:        create v locus of head_i splitting (q, γ]
15:        INSERT(q, tail, F)
16:        s_x(F) ← v
17:        find r = s_x(v) with FastFind
18:        p ← r
19:    end if
20:    update i
21: end while
22: mark terminal states
```

Výpis 3: Algoritmus přímé konstrukce $CDAWG$ v lineárním čase.

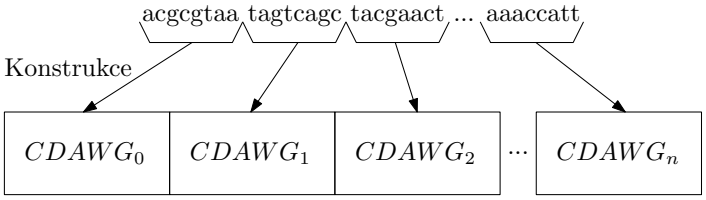
V obrázku 13 jsou zobrazeny jednotlivé kroky konstrukce $CDAWG(ababcb)$. Řetězec je zvolen stejný jako u příkladu konstrukce $DAWG(ababcb)$ 9 pro porovnání struktur. Již na první pohled je zřejmý nižší počet stavů u $CDAWG$, díky tomu je $CDAWG$ úspornější než $DAWG$.

4.3 Implementace

Nechť $S = s_0, s_1, \ldots, s_{|S|-1}$ je vstupní řetězec, jehož délka je $|S|$, a nechť L je kladné celé číslo, které určí maximální délku podřetězce. Řetězec S rozdělíme na $K = ceil(\frac{|S|}{L})$ podřetězců $X_0 \ldots X_{K-1}$, kde $|X_k| = L$. Poslední podřetězec X_{K-1} je délky $|X_{K-1}| = |S| - (K-1) * L; |X_{K-1}| \leq L$. Pro každý takový podřetězec je zkonstruován $CDAWG$ podle algoritmu Crochemore [6] nezávisle na ostatních podřetězcích. Výsledkem máme automaty



Obrázek 13: Přímá konstrukce kompaktního sufixového automatu $CDAWG(ababcb)$ podle algoritmu Crochemore. Jednotlivé kroky jsou popsány sufixem, který je v daném kroku přidán do $CDAWG$.



Obrázek 14: Ilustrace rozdělení řetězce S na podřetězce $X_0 \dots X_{K-1}$, pro které se následovně zkonstruuje $CDAWG$ nezávisle pro každý podřetězec. V tomto případě je délka rozdělení $L = 8$.

$CDAWG_0$ ... $CDAWG_{K-1}$. Takto zkonstruované automaty se zapíší na disk zvlášť; neprovádí se žádné sloučení struktur, jako tomu je například u algoritmu pro konstrukci sufixového stromu TRELLIS [20]. Při vyhledávání nad řetězcem S se tato datová struktura pouze načte z disku a automaty se z dat zpětně zkonstruují. Ilustrace rozdělení řetězce pro jednotlivé automaty je zobrazena na obrázku 14.

Dále mějme na vstupu řetězec X délky $|X|$, který chceme najít ve vstupním řetězci S a nezáporná čísla X_A, X_B, která představují indexy v řetězci S. Při vyhledávání řetězce X nás poté zajímají pouze výskyty, které se nachází mezi index X_A a $X_B - |X|$. Jestliže je $X_A = 0$, pak se řetězec S prohledává od $0 \dots X_B$. Obdobně, pokud je $X_B = 0$, pak se řetězec S prohledává od $X_A \dots |S|$. Počáteční index A a koncový index B automatů, které je potřeba projít pro nalezení řetězce X, lze určit jednoduše:

$$A = floor(\frac{X_A}{L})$$
$$B = floor(\frac{X_B - |X|}{L})$$

Znamená to, že pro známé hodnoty A a B je potřeba prohledat automaty $CDAWG_A, CDAWG_{A+1}, ..., CDAWG_B$. Pokud tedy pro nějaký řetězec X známe i X_A, X_B, dokážeme v konstantním čase určit automaty, které je potřeba prohledat. Další výhodou jsou efektivní možnosti paralelního zpracování, které jsou dále diskutovány v sekci 4.8. Detaily k vyhledávání řetězců jsou popsány v kapitole 4.7.

Tato implementace je cílená na hardware s více než 900 GB operační paměti. Pokud bychom vyhledávání nad dlouhým vstupním řetězcem chtěli pustit na klasických osobních počítačích, během okamžiku dojde paměť. Právě díky rozdělení automatů lze tento problém obejít do značné míry a to pomocí *lazy-loading*. Více je tomuto tématu věnováno v kapitole s experimenty 5.

Protože je $CDAWG$ acyklický automat, jeho hrany jsou orientované. Přechody mezi stavy jsou implementovány pomocí *adjacency lists*. To znamená, že každý stav si uchovává informace o sousedních stavech. V implementaci *adjacency lists* je přístup k jednotlivým sousedům pro daný stav v čase $\mathcal{O}(\log |\Sigma|)$, ale je potřebuje méně paměti. Alternativně lze použít matici sousednosti, ve které lze přistupovat k sousedům daného stavu v konstantním čase, ale potřebuje

```
#define INDEX_TYPE unsigned int

struct edge
{
   INDEX_TYPE length;
   INDEX_TYPE stateId;
};

struct state
{
   bool isAccepting;        // is state accepting; not written to disk
   INDEX_TYPE offset;
   // INDEX_TYPE length;    // used during construction; not written to disk
   // INDEX_TYPE previousId;  // suffix link; not written to disk
   DynArray<edge> *edges;
};
```

Výpis 4: Datové struktury pro hrany a stavy v $CDAWG$.

$\mathcal{O}(States(x) * |\Sigma|)$ [6] paměti. Ve výpise 4 je struktura datových typů *state* pro stavy a *edge* pro hrany. Každý stav obsahuje pole hran, které z daného stavu vedou. V hraně je uchována pouze informace cílového stavu a délka hrany.

4.4 Zápis na disk

Pro vyhledání řetězce X není potřeba znát všechny vlastnosti automatu, které jsou využity při jeho konstrukci. Strukturu stavů a hran si lze připomenout ve výpise 4. Zapisuje se počet stavů automatu, pro každý stav index *offset*, což je pozice ve vstupním řetězci S a počet hran, které z něj vedou. Každá hrana obsahuje index *stateId* stavu, do kterého daná hrana vede a délku řetězce *length*, který popisuje danou hranu. Řetězec e popisující danou hranu lze získat z informace $offset$ stavu, do kterého hrana vede: $offset = states[stateId].offset$, a délky hrany: $e = S_{offset-length...offset}$. Konečný stav automatu není potřeba zapisovat. Protože jsou sufixové odkazy použity pouze při konstrukci, rovněž se nezapisují. Formát, ve kterém se data $CDAWG$ zapisují na disk, je popsán ve výpise 5.

Díky rozdělení vstupního řetězce na podřetězce maximální délky L lze dále ušetřit místo výsledného automatu zapsaného na disk. Indexy ve struktuře pro stav a hranu jsou zároveň indexy ve vstupním řetězci pro daný automat, jsou implementovány na 4 byty. Pokud se sestaví $CDAWG$ pouze pro krátký podřetězec X, lze indexy v daném automatu chápat jako relativní pro řetězec daného automatu. V $CDAWG$ je uložen index ve vstupním řetězci S, na kterém daný automat začíná. Tyto indexy však můžeme chápat jako relativní pro daný automat. Protože známe počáteční index každého automatu ve vstupním řetězci, to je $I = N * L$ pro N-tý automat, při vhodně zvolené konstantě rozdělení L lze ušetřit místo na disku a indexovat méně než 4 byty.

```
 1: procedure WriteToBuffer(CDAWG)
 2:     write(number of states except sink)
 3:     // Don't write sink state
 4:     for all state in CDAWG do
 5:         if state == sink then
 6:             Continue
 7:         end if
 8:         offset ← value of state.offset with 1 bit set for state.isAccepting
 9:         write(offset)
10:         write(number of edges in state)
11:         for all edge in state do
12:             write(length of edge)
13:             write(state id of edge)
14:         end for
15:     end for
16: end procedure
17:
18: for all CDAWG in all built CDAWGs do
19:     WriteToBuffer(CDAWG)
20: end for
```

Výpis 5: Algoritmus zápisu $CDAWG$ na disk

V této implementaci lze pomocí 1 bytu indexovat automaty s konstantou $L \leq 128(2^7)$, pro 2 byty je to $L \leq 32768(2^{15})$, pro 4 byty je to $L \leq 2147483648(2^{31})$. Tyto maximální hodnoty jsou zmenšeny o 1 bit, protože při zápisu je 1 bit z proměnné *offset* všech stavů použit k zakódování hodnoty z boolean hodnoty *isAccepting*. Jinými slovy, 1 bit z *offset* určuje, zda je daný stav přijímající.

Při zápisu dat z každého automatu se současně vytváří pomocný index, který je popsán v následující sekci.

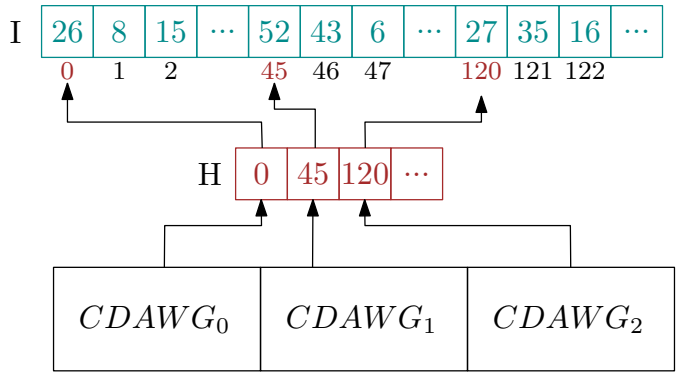
4.5 *Lazy-loading*

Při vyhledávání pomocí *lazy-loading* nejsou jednotlivé automaty předem inicializovány. Program v paměti uchovává data potřebná pro všechny automaty v jednom souvislém poli dat I. Pomocný indexový soubor H je rovněž načten do paměti jako souvislé pole dat. Pomocí tohoto pomocného indexu je implementováno *lazy-loading* dat při vyhledávání řetězců.

Pro i-tý automat $CDAWG_i$ se dohledá hodnota v pomocné indexové struktuře na pozici H_i. Tato hodnota určuje index v poli I na kterém bytu začínají data pro $CDAWG_i$. Následně lze $CDAWG_i$ inicializovat přečtením dat z $I[H_i]$. Provede se vyhledání v tomto automatu a následně je paměť uvolněna.

Podobným způsobem lze data pro automaty načítat přímo z disku, nemusí být neustále držena v paměti. Avšak v praxi to znamená značné zpomalení vyhledávání právě kvůli častým přístupům na disk. Na obrázku 15 je zobrazeno schéma, které popisuje využití této pomocné

indexové struktury pro *lazy-loading*. Například pro $CDAWG_1$ se zjistí hodnota z pole $H_i = 45$. Je to index, který říká, že data pro automat $CDAWG_1$ se nachází na indexu 45 v poli I. Hodnoty v poli I představují byty pro rekonstrukci jednotlivých automatů. Toto schéma slouží pouze pro ilustraci a nepředstavuje reálné hodnoty.



Obrázek 15: Ilustrace určení pozice pro data jednotlivých automatů využitím pomocné indexové struktury.

4.6 Komprese dat

Protože automat pracuje nad abecedou $|\Sigma| = 4$, lze jeden znak vyjádřit hodnotou, kterou lze reprezentovat pomocí dvou bitů. Do jednoho bytu se dají zapsat 4 znaky po 2 bitech. Tabulka 2 popisuje hodnoty jednotlivých znaků využité při kompresi.

Tabulka 2: Tabulka hodnot pro znaky abecedy při kompresi

Znak	A	C	G	T
Hodnota	0x00	0x01	0x02	0x03

Řetězec délky N znaků lze tímto způsobem uložit do paměti velikosti $C = ceil(\frac{N}{4})$ bytů. Pro zápis do souboru je dále potřeba znát, kolik znaků se v souboru po kompresi opravdu nachází. Tuto informaci lze například zjistit jako

$$\frac{filesize - 1}{4} - 4 + X$$

kde $X \in < 1, 4 >$ je počet znaků, které se nacházejí v posledním bytu. X se zapíše jako první byte v souboru.

Alternativní způsob zápisu, který byl zvolen při implementaci, je následující: prvních 8 bytů je pro C (velikost dat v bytech po kompresi), 1 byte pro X (počet znaků v posledním bytu) a zbytek jsou komprimovaná data. Touto cestou lze přečíst prvních 8 bytů ze souboru a ihned zjistit, kolik znaků se v souboru nachází, bez nutnosti zjištění velikosti souboru.

Při samotné konstrukci datové struktury či vyhledávání řetězce se komprese nevyužívá, protože je k dispozici dostatek volné paměti na cílovém hardware. To znamená, že po přečtení dat ze souboru se data rozbalí do paměti a dále se s nimi pracuje jako polem znaků po 1 bytu. Také to usnadní a urychlí náhodný přístup do pole, protože každé přečtení hodnoty znamená několik operací.

Algoritmus komprese do 2 bitů pracuje po 4 znacích. Ke zjištění hodnoty daného znaku je využitá vyhledávací tabulka 3. Tabulka obsahuje hodnoty určené v tabulce 2, které se nachází na indexech, které odpovídají danému znaku. Všechny ostatní prvky tabulky mají libovolnou hodnotu, protože nejsou využity.

Tabulka 3: Vyhledávací tabulka pro kompresi

Index	0	1	2	3	4	5	6	7	...	18	19
Znak	A	B	C	D	E	F	G	H	...	S	T
Hodnota	0x0	0xff	0x1	0xff	0xff	0xff	0x2	0xff		0xff	0x3

Pro převod zpátky z 2 bitové komprese na původní řetězec je využita vyhledávací tabulka, která odpovídá tabulce 2. Hodnota každých 2 bitů z daného bytu se použije jako index v této tabulce a získá se původní znak. Komprese i dekomprese si musí pro poslední byte ošetřit krajní případy podle hodnoty X, tedy počtu znaků v posledním bytu.

4.7 Vyhledávání řetězců

Nalezení prvního výskytu řetězce X v daném automatu je ekvivalentní průchodu automatu přes hrany, které odpovídají znakům v řetězci X, v čase $\mathcal{O}(|X|)$. Tento algoritmus je popsán ve výpise 6. Protože v této implementaci byl vstupní řetězec S rozdělen na podřetězce $X_0 \dots X_{K-1}$, může se stát, že hledaný řetězec se nebude nacházet celý v daném automatu, ale pouze nějaký prefix p. Proto jsou výsledky vyhledávání řetězce rozděleny do dvou kategorií: úplná shoda (celý řetězec byl nalezen v daném automatu) a částečná shoda v případě nalezení prefixu hledaného řetězce. Pro každou částečnou shodu je poté nutné porovnat zbývající znaky v hledaném řetězci. Složitost algoritmu samotného se nemění, je to pouze krajní případ, na který je potřeba myslet.

Samotným průchodem automatu lze nalézt pouze první výskyt hledaného řetězce v daném automatu. Pokud byla nalezena úplná shoda, jedná se o první výskyt řetězce. V případě, že nebyla nalezená částečná shoda, je pak potřeba zkontrolovat všechny nalezené částečné shody, zda jsou platnými podřetězci vyhledávaného řetězce X a vybrat nejmenší index takového výskytu.

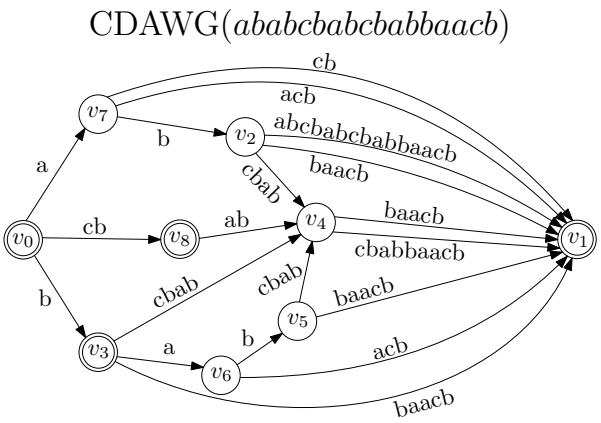
Pro nalezení všech výskytů řetězce je potřeba zjistit všechny možné cesty, které vedou ze stavu, ve kterém byl vyhledávaný řetězec nalezen, do stavu *sink* daného automatu. Každá taková nalezená cesta představuje další výskyt hledaného řetězce. Realizace tohoto kroku je v algoritmu 6 na řádku 16). Tento krok vyhledání cest do přijímacího stavu není potřeba provádět při každém vyhledávání. Pro každý stav automatu lze zjistit délky všech cest vedoucí do přijímacího stavu a pro daný stav si tyto délky zapamatovat. Pokud známe všechny takové cesty, které z daného

stavu vedou do přijímacího, kde c je počet takových cest, dokážeme nalézt všechny výskyty daného řetězce v rámci jednoho automatu v čase $\mathcal{O}(|X| + c)$. Nevýhodou je větší paměťová náročnost, kde každý stav si uchovává informaci o délkách všech možných cest do *sink* stavu.

```
1: procedure FindAll(string, state_in, matched)
2:     edge ← first edge from state_in
3:     while edge is valid do
4:         S ← target state from edge
5:         offset ← offset of S
6:         if S is accepting then
7:             // A zero-length edge leads to final state from here
8:             AddPartialMatch(offset − matched)
9:             edge ← next edge
10:            continue
11:        end if
12:        if edge contains string then
13:            j ← count of matched characters
14:            if entire string was found then
15:                AddFullMatch(offset − matched − j)
16:                FindAllPaths(S)
17:                exit
18:            else if reached end of automaton then
19:                AddPartialMatch(offset − matched − j)
20:                exit
21:            else
22:                string ← substring(string, j, len(string) - j).
23:                // Recursively call this function
24:                FindAll(string, S, matched + j)
25:                exit
26:            end if
27:        end if
28:        edge ← next edge
29:    end while
30: end procedure
31: // Begin searching
32: FindAll(searched_string, first_state, 0)
```

Výpis 6: Algoritmus nalezení všech výskytů řetězce

Následuje několik příkladů pro vyhledávání v řetězci $S = ababcbabcbabbaacb$. Indexy jed-



Obrázek 16: $CDAWG(ababcbabcbabbaacb)$

notlivých znaků řetězce S jsou v tabulce 4. Vyhledávaným řetězcem je $X = cbab$, postup pro vyhledání se řídí algoritmem ve výpise 6. Přechody mezi stavy při vyhledávání X je zobrazen v tabulce 5. První řádek je pouze informativní a slouží k inicializaci hodnoty Y a počátečního stavu. T_A a T_B označují výchozí a cílový stav pro daný krok, E je řetězec, který popisuje hranu z T_A do T_B délky $|E|$, Y je řetězec, který zbývá dohledat a O je počet znaků, které zbyly při vyhledávání, tedy $O = |E| - |Y|$. Pro ilustraci je zkonstruovaný $CDAWG(ababcbabcbabbaacb)$ zobrazen v obrázku 16.

Tabulka 4: Tabulka indexů pro řetězec $S = ababcbabcbabbaacb$, pro které se sestaví jeden $CDAWG$.

$CDAWG_0$	a	b	a	b	c	b	a	b	c	b	a	b	b	a	a	c	b
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Tabulka 5: Tabulka přechodů mezi stavy při vyhledávání řetězce $X = cbab$ v automatu $CDAWG_0(ababcbabcbabbaacb)$

T_A	T_B	E	$\|E\|$	Y	O
-	v_0	-	-	$cbab$	0
v_0	v_8	cb	2	ab	0
v_8	v_4	ab	2	$-$	0

Z tabulky 5 vyplývá, že vyhledávání končí po dvou cyklech. Algoritmus skončil ve stavu v_4 s hodnotou $offset = 8$. Pro zjištění indexu prvního výskytu $X = cbab$ v $S = ababcbabcbabbaacb$ lze definovat rovnici:

$$i_{first} = offset(T) - |X| - O$$

kde $offset(T)$ je $offset$ hodnota stavu, ve kterém vyhledávání skončilo, $|X|$ délka vyhledávaného řetězce. Hodnota O byla definována výše. Aplikováním vzorce získáme hodnotu $i_{first} = 8 - 4 - 0 = 4$, tedy první výskyt řetězce $X = cbab$ v řetězci $S = ababcbabcbabbaacb$ se nachází na indexu 4. Výsledek lze potvrdit v tabulce 4.

Pro dohledání všech výskytů X musíme projít celý $CDAWG$ ze stavu, ve kterém algoritmus skončil, a zjistit všechny cesty do *sink* stavu, který je v obrázku 16 zobrazen jako v_1. V tomto případě ze stavu v_4 vedou právě dvě hrany do *sink* stavu, ale ve složitějším řetězci může takových cest existovat mnoho. Aplikováním následující rovnice pro každou takovou cestu lze získat indexy všech výskytů X:

$$i_j = offset(T) - |X| - O - |P_j|$$

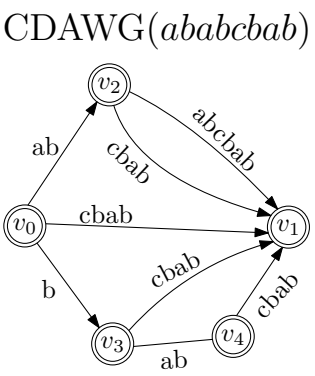
kde hodnota $offset$ pro *sink* stav je vždy rovna délce řetězce, nad kterým je daný automat sestaven, P_j je délka j-té cesty do *sink* stavu. Konkrétně $i_0 = 17 - 4 - 0 - 9 = 4$ a $i_1 = 17 - 4 - 0 - 5 = 8$. Lze si povšimnout, že i_0 je duplikát prvního výskytu X na pozici i.

V těchto ukázkových situacích skončilo vyhledávání na konci nějaké hrany. Uveďme ještě příklad pro vyhledání řetězce $X = a$. Tabulka přechodů 6 je v tomto případě triviální, ihned v prvním kroku je nalezen znak c, avšak změní se hodnota $O = 1$. Nalezením všech cest do *sink* stavu a aplikováním výše uvedeného vzorce získáme indexy $i_{first} = 4$, $i_0 = 4$, $i_1 = 8$ a $i_2 = 15$.

Tabulka 6: Tabulka přechodů mezi stavy při vyhledávání řetězce $X = c$ v automatu $CDAWG_0(ababcbabcbabbaacb)$

T_A	T_B	E	$\|E\|$	Y	O
-	v_0	-	-	c	0
v_0	v_8	cb	2	–	1

Nyní vezměme v úvahu případ, kde se řetězec $S = ababcbabcbabbaacb$ rozdělí podle $L = 8$, tedy na $K = 3$ částí. Tabulka 7 názorně ukazuje takové rozdělení. Vyhledávaný řetězec je tentokrát $X = abcabc$. Vizuálně vidíme, že počátek všech výskytů X se nachází v automatu $CDAWG_0$, algoritmus by samozřejmě prošel i automat $CDAWG_1$. Pro názornost je ilustrováno pouze vyhledávání v automatu $CDAWG_0$. Automat $CDAWG_2$ nebude algoritmem vůbec navštíven, protože je zřejmé, že se řetězec délky $|X| = 6$ nemůže nacházet v automatu, který zpracovává řetězec délky 1. Přechody mezi stavy při vyhledávání X je zobrazen v tabulce 8. Automat $CDAWG_0$ je zkonstruován v obrázku 17.



Obrázek 17: $CDAWG(ababcbab)$

Tabulka 7: Tabulka rozdělení řetězce $S = ababcbabcbabbaacb$ na $K = 3$ části. Pro každý takový podřetězec je sestrojen $CDAWG$. Indexy jednotlivých znaků jsou relativní k patřičnému automatu.

$CDAWG_0$	a	b	a	b	c	b	a	b
	0	1	2	3	4	5	6	7
$CDAWG_1$	c	b	a	b	b	a	a	c
	0	1	2	3	4	5	6	7
$CDAWG_2$	b							
	0							

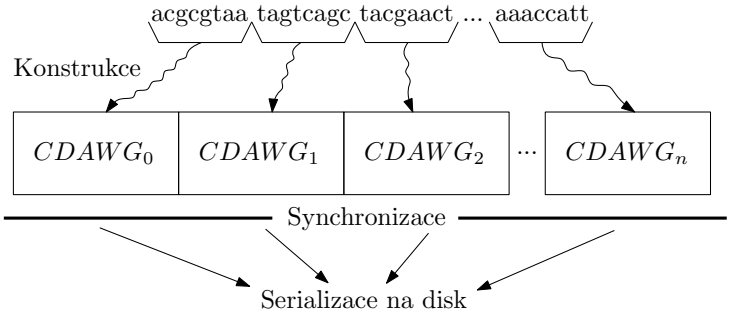
Tabulka 8: Tabulka přechodů při vyhledávání řetězce $X = abcbabc$ v automatu $CDAWG_0(ababcbab)$

T_A	T_B	E	$\|E\|$	Y	O	Matched	Accepting
-	v_0	-	-	$abcbabc$	0	0	true
v_0	v_2	ab	2	$cbabc$	0	2	true
v_2	v_1	$cbab$	4	c	0	6	true

V tabulce 8 přibyly dva sloupce $Matched$, tedy počet znaků doposud nalezených a sloupec *Accepting*, který říká, zda je cílový stav T_B pro daný krok přijímající, tedy pokud stav obsahuje prázdnou hranu do *sink* stavu. Protože byl *Accepting* stav nalezen zatímco celý řetězec X zatím nebyl dosud nalezen, zapamatuje se částečná shoda na indexu

$$i = offset(T) - Matched$$

První částečná shoda je tedy nalezena na indexu $i_0 = 8 - 2 = 6$. Pro i_0 je zbývající hledaný řetězec $Y = cbabc$ a pokračuje se ve vyhledávání od posledního indexu pro daný $CDAWG$,



Obrázek 18: Schéma návrhu paralelní konstrukce automatů. Pro každý automat je vytvořeno vlákno, ve kterém se zkonstruuje. Poté se vlákna synchronizují a sériově se automaty zapíší na disk.

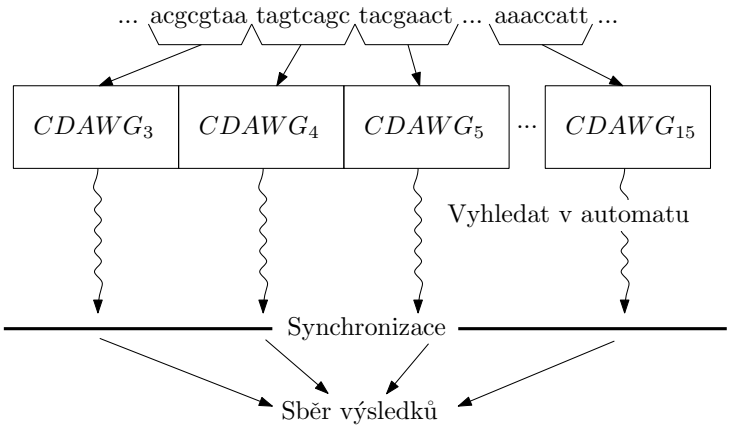
v tomto případě od indexu 8. Nyní již jde o jednoduché porovnání znak po znaku se vstupním řetězcem. Vyhledávání v tomto případě selže na indexu 12, tedy na posledním znaku v X.

Dalším krokem v algoritmu se skončilo v *sink* stavu v_1, není již kam dále se přesunout. Je tedy přidána další částečná shoda na indexu $i_1 = 8 - 6 = 2$. Opakuje se stejný postup jako u předchozí částečné shody, avšak toto vyhledávání skončí na indexu 8 úspěšně.

4.8 Možnosti paralelního zpracování

Paralelní zpracování není součástí této práce, proto jsou zde pouze zmíněny teoretické možnosti paralelního zpracování. Konstrukce automatů pro každý podřetězec $X_0 \dots X_{K-1}$ je nezávislá na ostatních automatech, díky tomu lze algoritmus konstrukce spustit v K vláknech na procesoru na způsob rozděl a panuj. V průběhu konstrukce automatu lze sledovat aktuální velikost v bytech, kterou automat bude potřebovat při zápisu na disk. Po konstrukci všech automatů se alokuje potřebná paměť na základě sumy těchto sledovaných velikostí všech automatů. Jakmile všechny automaty zapíší potřebné hodnoty, data se uloží na disk jako jeden zápis. Na obrázku 18 je schéma návrhu paralelní konstrukce automatů.

Obdobně může pracovat i algoritmus pro vyhledání řetězce. Protože pro vyhledávaný řetězec X a jeho počáteční a konečné pozice X_A, X_B známe automaty $C_A, C_{A+1}, \dots, C_B$, které je potřeba prohledat. Prohledání každého takového automatu lze spustit ve vlastním vlákně na procesoru. Pokud mámě na vstupu více řetězců, které potřebujeme vyhledat, obdobně můžeme spustit vyhledávání i pro ostatní takové řetězce. Na obrázku 19 je schéma návrhu paralelního vyhledávání v automatech.



Obrázek 19: Schéma návrhu paralelního vyhledávání v automatech. V této ilustraci předpokládejme délku rozdělení řetězce $K = 8$, indexy $A = 30$ a $B = 100$ pro vyhledávání řetězce. Automaty, které se musí prohledat, jsou tedy $CDAWG_3, CDAWG_4, \ldots CDAWG_{15}$. Nad každým automatem se spustí vyhledávání v unikátním vlákně. Poté se vlákna synchronizují a posbírají se výsledky z prohledávání jednotlivých automatů.

5 Experimenty

Po realizaci návrhu implementace bylo dalším krokem provézt několik experimentů a určit, jak výhodná je metoda rozdělení vstupního řetězce. V této kapitole jsou popsány následující experimenty na datech *data64mb*: rychlost konstrukce automatů, velikost těchto automatů na disku, rychlost vyhledávaní řetězců nad automaty, rychlost vyhledávání delších dotazů, rychlost nalezení prvního výskytu řetězce a rychlost vyhledávání pomocí *lazy-loading*. Dále je zde srovnání těchto měření při vhodné délce rozdělení oproti měření na jednomu automatu. Na závěr byly provedeny výše zmíněné experimenty na datech *human*, které obsahují 2,6 miliard znaků.

Experimenty probíhaly na zařízení s operačním systémem SUSE Linux Enterprise Server 12, 84 procesorů Intel(R) Xeon(R) CPU E5-4610 0 @ 2.40GHz, 978 GB RAM. Data byla zapisována na disky Hitachi HUC109030CSS600. Programy byly překládány překladačem GNU C++ (SUSE Linux) 4.8.5 s optimalizací *-O3*.

5.1 Vstupní data

Data pro experimenty nad implementací $CDAWG$ byla reálná data genetických sekvencí. Protože je implementace cílená pro abecedu $\Sigma = a, c, g, t$, testovací data obsahovala pouze znaky z této abecedy. Dotazy nad těmito daty byly vygenerovány programem, který ze vstupního řetězce vybral podřetězec na náhodné pozici a zapsal jej ve formátu, který program pro vyhledávání nad $CDAWG$ očekává na vstupu: *start, end, length, string*. Hodnoty *start* a *end* určují právě pozice X_A a X_B ve vstupním textu, kde by se měl řetězec *string* délky *length* vyhledávat. Program je dále řízen několika parametry: *min, max, num offset*, kde *min* a *max* je minimální a maximální délka podřetězce (délka se náhodně vybere v tomto rozsahu), *num* je počet vygenerovaných podřetězců. Protože program přesně ví, na které pozici se daný podřetězec nachází, je rozsah pozic *start* a *end* zvětšen v obou směrech právě o hodnotu *offset*. Vyhledávání podřetězce pak probíhá i v okolí výskytu hledaného podřetězce a často se stává, že je nalezen více než jeden výskyt. Program pro vyhledávání pak na vstupu očekává cestu k souboru, který obsahuje dotazy v tomto formátu. Příklad takto vygenerovaných dotazů je v tabulce 9. Řetězec s indexem 0 se má vyhledávat ve vstupním řetězci mezi pozicemi 7426 a 17447. Protože je $offset = 5000$, reálně se tento řetězec nachází na indexu $X_A + offset = 7426 + 5000 = 12426$. Tato informace samozřejmě není známa programu pro vyhledávání.

V tabulce 10 je výpis vstupních dat, která byla testována. Velikost abecedy je u všech dat stejná $|\Sigma| = 4$. Data *human* je 2,6 GB veliký řetězec popisující lidskou DNA. Náhodným výběrem 64 miliónů dlouhé sekvence znaků z dat *human* vznikla data *data64mb* pro účely jednodušších experimentů.

Tabulka 9: Příklad vygenerované sady dotazů s parametry $num = 5$, $min = 10$, $max = 50$, $offset = 5000$.

index	X_A	X_B	*length*	*string*
0	7426	17447	21	aagttctacagttcacttgaa
1	10853	20872	19	ggtgcctgtattcccagct
2	25314	35328	14	ccatccttaccatg
3	22365	32396	31	tgtatctatttcattgttgtgtcctattagt
4	16163	26193	30	agctcatcagactaacagcagacttctcag

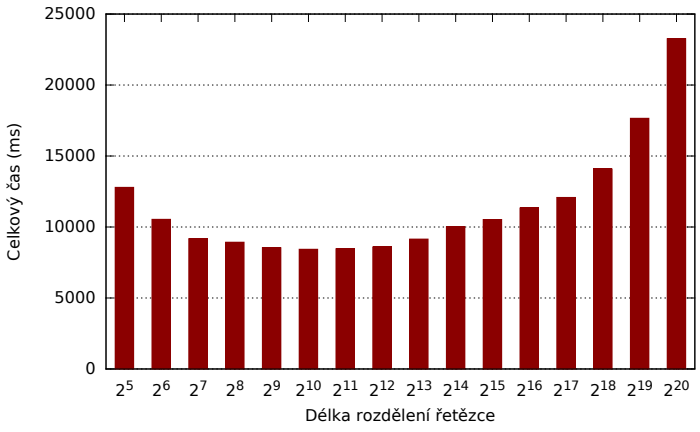
Tabulka 10: Vstupní soubory pro experimenty

Soubor	$\|S\|$
data64mb	63 944 257 ≈ 64 M
human	2 745 186 602 ≈ 2,6 G

5.2 Čas pro konstrukci automatů

Různé délky rozdělení vstupního řetězce vytvoří různě veliké automaty, proto se tento experiment zaměří na celkový čas konstrukce všech automatů. Součástí měření není zápis výsledných dat na disk, je měřen pouze čas konstrukce automatů a zápis do jednotného bufferu v paměti. Měření proběhlo nad rozdělením od 2^5 do 2^{20} znaků, viz graf 20. Na ose X v grafech je délka rozdělení řetězce, osa Y popisuje čas konstrukce všech automatů pro dané rozdělení.

Pokud by bylo požadavkem zkonstruovat indexovou strukturu za běhu programu a opětovně nalézt malé množství řetězců, dalo by se uvažovat o využití rozdělení kolem 2^{10} znaků. Pro obsáhlejší dotazy, jako jsou právě testované sady dotazů obsahující 10 000 dotazů, se již nevyplatí zaměřit se na rychlejší konstrukci, ale na rychlejší vyhledávání.



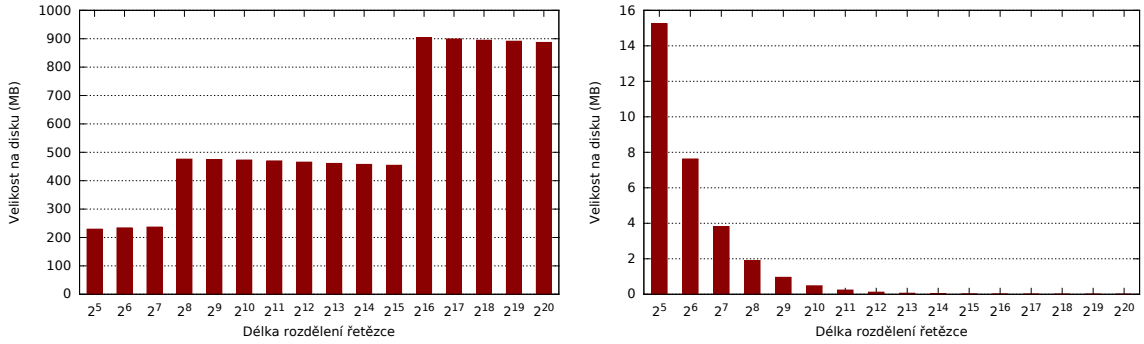
Obrázek 20: Graf porovnávající celkový čas potřebný pro konstrukci všech automatů pro dané rozdělení pro data *data*64*mb*.

5.3 Velikosti automatů

Dalším krokem bylo zjistit, jaká délka rozdělení je ideální z pohledu místa potřebného k zápisu automatů na disk. Velikosti rozdělení byly zvoleny od 2^5 do 2^{20} znaků, stejně jako při měření časů pro konstrukci automatů. Výsledky tohoto měření jsou v levém grafu v obrázku 21, který zobrazuje velikost souboru obsahující data zkonstruovaných automatů při daném rozdělení. Na ose X v grafech je délka rozdělení řetězce, osa Y popisuje velikost na disku v MB.

Pro velikosti rozdělení $K \leq 128(2^7)$ lze zapisovat každý index v automatu na 1 byte, pro $K \leq 32768(2^{15})$ se indexy vlezou do 2 bytů a pro vyšší hodnoty vystačí 4 byty. Proto jsou v levém grafu v obrázku 21 viditelné dva "schody". Konkrétně mezi hodnotami 2^7 a 2^8, protože se automaty s rozdělením pod 2^8 znaků včetně indexují na 1 byte, pro více než 128 znaků je potřeba použít 2 byty. Podobně tomu je mezi hodnotami 2^{15} a výše, kde je přechod z indexování se 2 byty na 4 byty. Detaily o možnostech zápisu na disk a vysvětlení, proč byly zvoleny zrovna hodnoty 2^7 a 2^{15}, jsou probrány v kapitole 4.4.

Dále je v pravém grafu v obrázku 21 zobrazeno porovnání velikostí pomocného indexu, které je potřeba pro *lazy-loading* automatů v paměti. Velikost na disku začíná na $15MB$ pro délku rozdělení 32 znaků a rapidně klesá s delšími řetězci, proto lze velikost této struktury zanedbat v porovnání s velikostí souboru všech automatů.



Obrázek 21: Vlevo graf velikostí automatů uložených na disku na datech *data64mb*. Vpravo graf velikostí pomocného indexu potřebného pro *lazy-loading* v paměti na datech *data64mb*.

Z grafů lze vypozorovat, že velikosti automatů na disku mezi různými rozděleními se příliš neliší. Rozhodující je velikost indexování, která byla diskutována výše. V případě, kdy je žádoucí zhruba poloviční úspora při zápisu na disk, lze zvolit rozdělení na 2^{15} znaků, protože lze automaty v tomto rozdělení indexovat pomocí 2 bytů, viz výše. Nižší délky rozdělení nelze doporučit kvůli výsledkům následujících experimentů.

5.4 Průměrný čas vyhledání řetězce

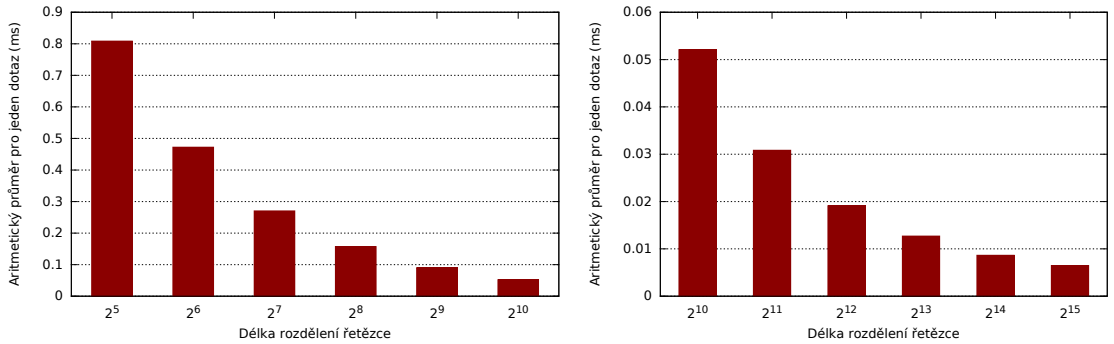
Pro měření průměrného času vyhledání řetězce jsou nejprve všechny automaty v paměti inicializovány, až pak je spuštěno vyhledávání a měření času. V těchto experimentech není využito

lazy-loading, které bylo testováno a popsáno v experimentu 5.6.

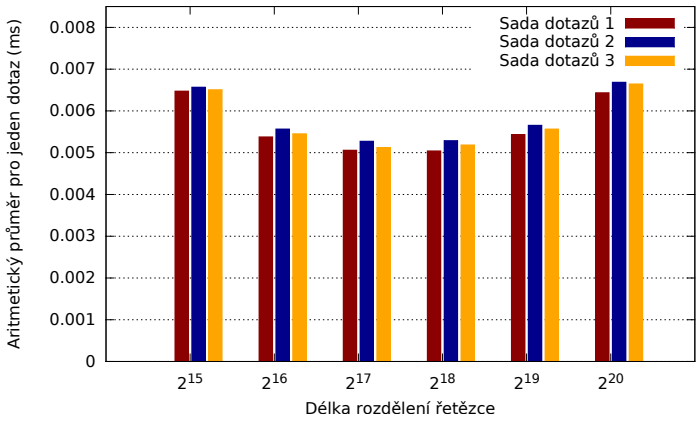
Následující experimenty využívají vygenerovanou sadu 10 000 dotazů, která byla popsána v úvodu této kapitoly. Pro každý dotaz v této sadě je změřen čas při vyhledávání. Vyhledávají se všechny výskyty řetězce mezi pozicemi *start* a *end* určenými pro daný řetězec. Pro všechny tyto naměřené časy se zjistí aritmetický průměr vyhledání jednoho dotazu. V následujících grafech jsou zobrazeny výsledků takto získaných měření s různými délkami rozdělení. Na ose X v grafech je délka rozdělení řetězce, osa Y popisuje průměrný čas v milisekundách.

První pokusy s měřením rychlosti vyhledávání všech výskytů hledaných řetězců proběhly pro rozdělení na velikosti od 2^5 znaků do 2^{10} znaků na datech *data*64*mb*. Výsledky těchto měření jsou v levém grafu v obrázku 22. Pro další experimenty byla zvolena velikost rozdělení od 2^{10} do 2^{15} znaků, výsledky zobrazeny na pravém grafu v obrázku 22. Dále se změřily časy pro rozdělení od 2^{15} do 2^{20} znaků, výsledky v grafu 23. Ve stejném grafu jsou rovněž zobrazeny časy pro dvě nově vygenerované sady dotazů se stejnými parametry, aby bylo zřejmé, že naměřené časy nejsou výhodné pro konkrétní sadu dotazů. Rychlost vyhledávání by se mohla razantně lišit v případě, že zvětšíme okolí vyhledávání řetězce, tedy *offset* parametr, či délku vyhledávaných řetězců. *Sada dotazů 1* je původní dotaz společný pro ostatní měření, zbývající dva jsou nově vygenerované dotazy se stejnými parametry.

V obrázku 22 lze z grafu vlevo vyčíst, že se z hlediska rychlosti vyhledávání vůbec nevyplatí rozdělit řetězce na takto krátké podřetězce. Na pravém grafu se časy dále snižují. Nejlepší výsledky lze vypozorovat v grafu 23, kde se jako nejrychlejší jeví vyhledávání s rozdělením mezi 2^{17} a 2^{18}. Zároveň se vyplatí si všimnout výsledků pro rozdělení 2^{15}, které je zhruba 2x efektivnější z hlediska místa na disku, viz předchozí experimenty s velikostí na disku. Při rozdělení na delší řetězce se časy navyšují, tato rozdělení nebyla prozkoumána.



Obrázek 22: Vlevo graf průměrných časů pro *data64mb* s rozdělením od 2^5 do 2^{10} znaků. Vpravo graf průměrných časů pro *data64mb* s rozdělením od 2^{10} znaků po 2^{15} znaků.

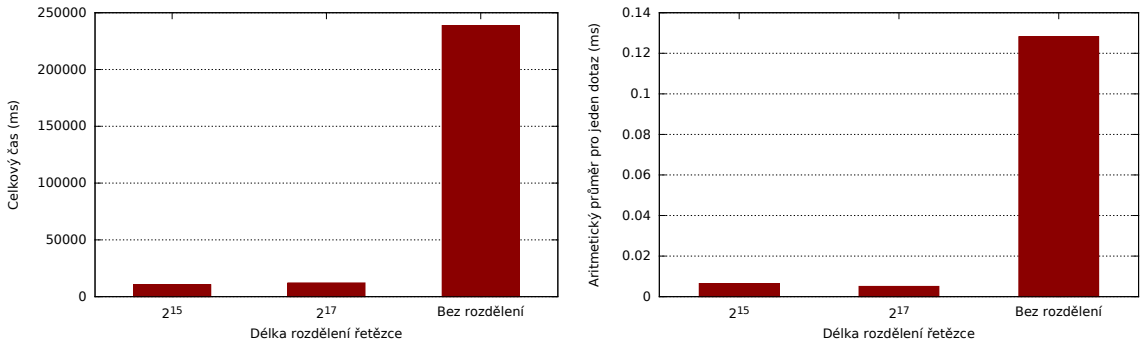


Obrázek 23: Graf průměrných časů pro *data*64*mb* se 3 sadami dotazů s rozdělením od 2^{15} do 2^{20} znaků.

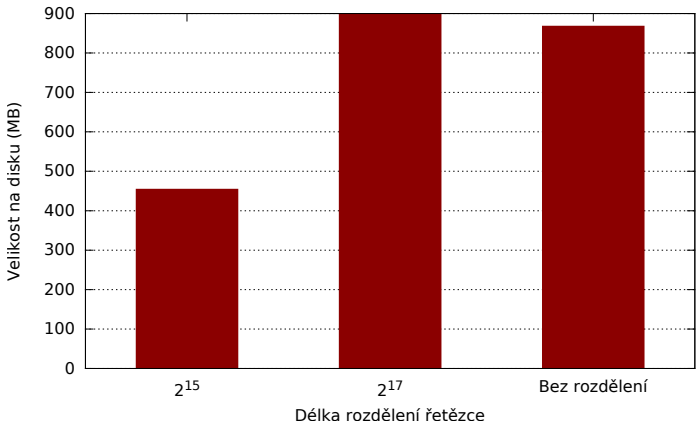
5.5 Vyhledávání v jediném automatu

V tomto experimentu se nad celým vstupním řetězcem zkonstruuje jeden $CDAWG$ nad daty *data*64*mb*. Je změřen čas konstrukce a průměrný čas vyhledání dotazu v grafu 24 a velikost automatu na disku v grafu 25, stejně jako u předchozích experimentů. V grafech jsou zároveň zaneseny výsledky pro rozdělení na 2^{15} a 2^{17} znaků, aby bylo možné porovnat naměřené hodnoty s nejlepšími výsledky při rozdělování vstupního řetězce.

V rychlosti konstrukce naprosto suverénně vítězí rozdělení na 2^{15} a 2^{17} znaků, stejně jako v rychlosti vyhledávání řetězců. Podle výsledků z experimentů času konstrukce automatu by zvítězilo rozdělení na 2^{10} znaků, ale vyhledávání je nad těmito automaty mnohem pomalejší, proto nebylo toto rozdělení zkoumáno. Rozdíl velikosti na disku je v podstatě zanedbatelný pro rozdělení 2^{17} znaků a bez rozdělení. Je zřejmé, že metoda rozdělení vstupního řetězce na vhodně dlouhé podřetězce, které se nezávisle zpracují, je výhodná ze všech hledisek při vyhledávání všech výskytů řetězce.



Obrázek 24: Vlevo graf srovnání času pro konstrukci automatů. Vpravo srovnání průměrného času pro vyhledání jednoho dotazu. V grafech je rozdělení 2^{15}, 2^{17} a bez rozdělení (jeden automat).



Obrázek 25: Graf velikostí automatů na disku při rozdělení 2^{15}, 2^{17} znaků a bez rozdělení (jeden automat).

5.6 *Lazy-loading* při vyhledávání

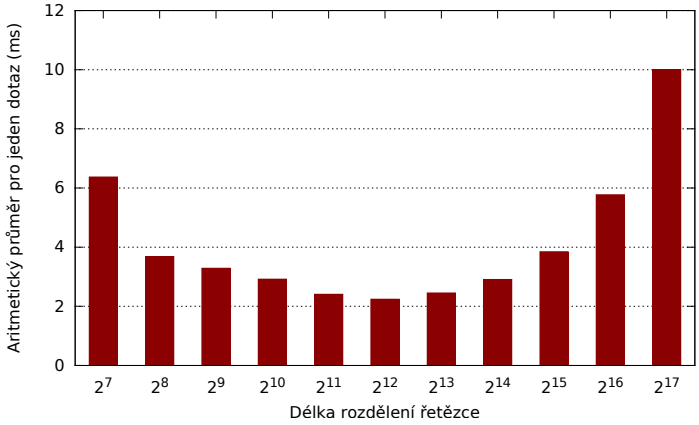
Doposud všechna měření probíhala až po přečtení dat z disku a inicializaci všech automatů. Slabší zařízení nemusí mít dostatek paměti pro udržení všech automatů. V tom případě lze použít *lazy-loading*. Při následujícím experimentu jsou sice nejprve data přečtena z disku, ale automaty nejsou inicializovány. Data jsou držena v paměti a automaty jsou zpětně zkonstruovány z těchto dat pouze v moment, kdy jsou potřeba. Následně po vykonání svého účelu je paměť pro daný automat uvolněna. Součástí měření v tomto experimentu je tedy i inicializace automatů v paměti. V praxi lze použít stejný princip s tím rozdílem, že data jsou čtena přímo z disku, čímž se ušetří další paměť. Způsobí to však značné zpomalení kvůli častým přístupům na disk. To však není součástí následujícího experimentu, zde jsou všechna data již v paměti.

Na grafu 26 jsou měření vyhledávání s využitím *lazy-loading*. V grafu nelze zároveň zobrazit výsledky z klasického vyhledávání, protože je *lazy-loading* mnohem pomalejší. Pro orientaci, vyhledávání s dopřednou inicializací s rozdělením 2^{17} trvalo v průměru 0.005 ms. V porovnání s nejlepším výsledkem s *lazy-loading* při rozdělení 2^{12}, které trvalo v průměru 2.3 ms.

V situaci, kdy je potřeba spustit program, načíst data a dohledat pouze jeden řetězec, či nízký počet řetězců, lze ospravedlnit využití *lazy-loading* s rozdělením na 2^{12} znaků. Avšak pro vyhledání mnoha řetězců, jak je tomu v těchto experimentech, je *lazy-loading* příliš pomalé. Pokud nejsou na vstupu známy indexy, mezi kterými se má vyhledávaný řetězec hledat, ztrácí tato metoda veškerý smysl. Tyto indexy jsou pro *lazy-loading* naprosto zásadní, bez nich by program dokola načítal všechny automaty.

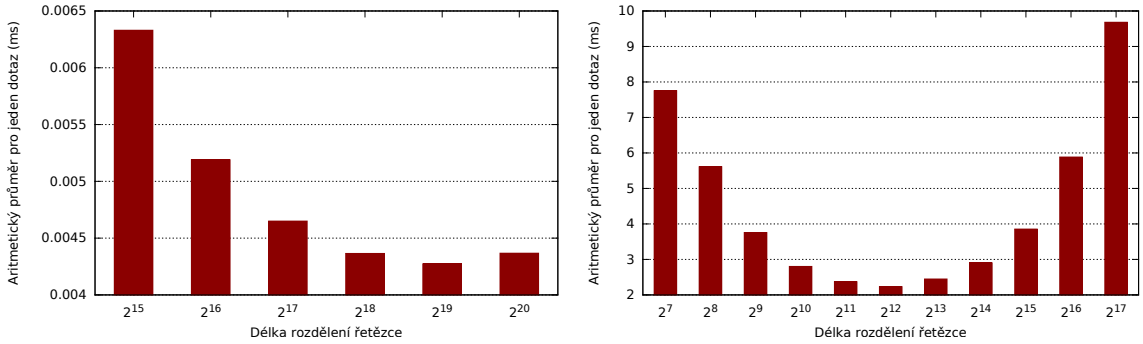
5.7 Vyhledávání delších řetězců

Doposud experimenty vyhledávání probíhaly nad řetězci délky 10 až 30 znaků. Následující experiment vyhledával řetězce délky 100 až 500 znaků, počet dotazů je stejný, tedy 10 000. Na



Obrázek 26: Graf rychlosti vyhledávání s využitím *lazy-loading* pro rozdělení od 2^7 do 2^{17}.

levém grafu v obrázku 27 jsou průměrné časy vyhledání dotazu, v pravém grafu je při stejném vyhledávání použito *lazy-loading*.

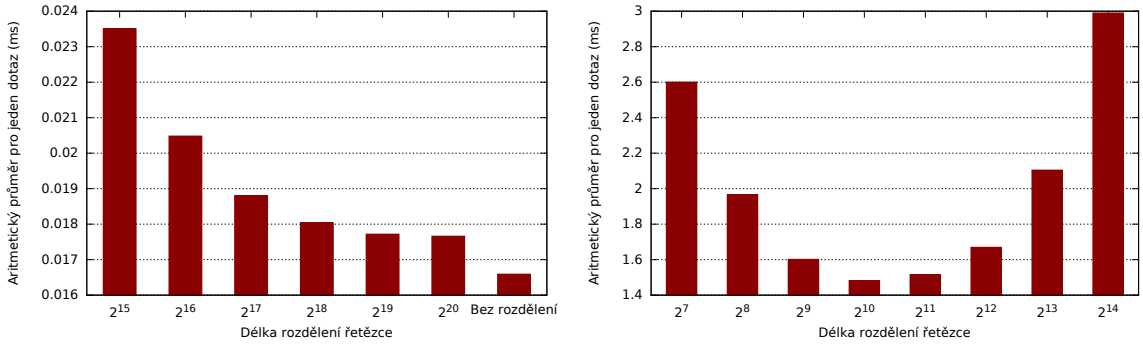


Obrázek 27: Vlevo graf srovnávající průměrný čas vyhledání řetězců délky 100 až 500 znaků nad daty *data64mb*. Pravý graf zobrazuje stejné vyhledávání s využitím *lazy-loading*.

Pokud jsou tyto výsledky porovnány s vyhledáváním kratších řetězců délky 10 až 30 znaků, viz grafy 23 a 26, vyhledávání s *lazy-loading* je opět nejvýhodnější při rozdělení na 2^{12} znaků. V klasickém vyhledávání bez *lazy-loading* se jeví jako ideální rozdělení na 2^{19} znaků. Ideální délka rozdělení je tedy různá podle délky řetězců, které jsou vyhledávány.

5.8 Nalezení prvního výskytu řetězce

V tomto experimentu byly změřeny průměrné časy pro vyhledání prvního výskytu hledaného řetězce. Byla použita původní sada dotazů, tedy řetězce délky 10 až 30 znaků. V levém grafu v obrázku 28 jsou průměrné časy nalezení prvního výskytu, na pravém grafu je stejné vyhledávání s využitím *lazy-loading*.



Obrázek 28: Vlevo graf srovnávající průměrný čas vyhledání prvního výskytu řetězců délky 10 až 30 znaků nad daty *data64mb*. Pravý graf zobrazuje stejné vyhledávání s využitím *lazy-loading*.

Jestliže se hledaný řetězec vyskytuje v automatu, existuje pro něj cesta v automatu. V případě jediného automatu je to operace $O(|X|)$, při rozdělení na podřetězce je složitost $O(K \cdot |X|)$, kde K je počet automatů zkonstruovaných nad vstupním řetězcem S. Dává tedy smysl, že nejrychlejší vyhledání prvního výskytu řetězce bude v případě jednoho automatu, jak levý graf v 28 dokazuje. Rovněž je zajímavé, že při využití *lazy-loading* se rozdělení 2^{10} jeví jako nejrychlejší v tomto experimentu, zatímco v předchozích experimentech při vyhledávání s *lazy-loading* bylo nejrychlejší rozdělení 2^{12}.

5.9 Experimenty na *human* datech

Závěrem byly provedeny experimenty měření velikosti automatů, času pro jejich konstrukci a průměrný čas vyhledávání na datech *human*, která obsahují 2,6 miliard znaků. Délky rozdělení byly zvoleny jako 2^7 pro testování při zápisu na 1 byte, dále 2^{12}, 2^{15} pro zápis na 2 byty a rozdělení 2^{16}, 2^{17} pro 4 byty. Další délky rozdělení nebyly testovány kvůli velikosti automatů na disku a jak časové, tak paměťové náročnosti konstrukce automatů pro tento vstup. Výsledky měření jsou popsány v tabulce 11.

Tabulka 11: Experimenty na *human* datech.

Délka rozdělení	Počet automatů	Velikost na disku (MB)
2^7	21 446 771	10 169.2
2^{12}	670 211	20 364.2
2^{15}	83 777	20 027.5
2^{16}	41 889	39 840.6
2^{17}	20 945	39 616.4
Délka rozdělení	**Čas konstrukce automatů (ms)**	**Průměrný čas vyhledávání 1 dotazu (ms)**
2^7	404 130	6.19493
2^{12}	377 300	4.1614
2^{15}	460 641	4.34121
2^{16}	508 286	8.58659
2^{17}	553 893	12.172

Kvůli nedostatku v implementaci, kde jeden objekt automatu $CDAWG$ zabírá příliš mnoho paměti v programu, bylo potřeba tento experiment spustit s využitím *lazy-loading*. Rozdělení na 2^{12} znaků vykazuje nejlepší rychlost vyhledávání, následuje tak trend předchozích experimentů s *lazy-loading* na *data64mb* datech. Spolu s výhodným zápisem na disk pomocí 2 bytů, kdy celková velikost automatů je zhruba $\approx 20GB$, je toto rozdělení výhodné pro vyhledávání s *lazy-loading*. Pro zápis automatů na disk je pro rozdělení 2^{15} potřeba zhruba 7,6 bytů na každý znak vstupu, zatímco při rozdělení 2^{17} je potřeba zhruba 15,1 bytů na znak. Čas konstrukce automatů rovněž odpovídá předchozím experimentům, kde bylo nejrychlejší rozdělení kolem 2^{10} znaků s využitím *lazy-loading*.

Přestože byl tento experiment proveden pomocí *lazy-loading*, pointou je, že se pořád vyplatí vstupní řetězec rozdělovat. Zkonstruovat pouze jeden $CDAWG$ nad celým vstupem *human* nebylo možné, konstrukce trvala několik hodin a nakonec došla paměť kvůli rekurzi. Díky rozdělení vstupu lze data *human* indexovat v rozumném čase a rovněž v něm i rychle vyhledávat.

6 Závěr

Úspěšně byl realizován návrh a implementace kompaktních sufixových automatů. Vstupní řetězec se rozdělí na několik podřetězců, kde každý takový podřetězec je dlouhý K znaků a pro každý podřetězec je nezávisle zkonstruován kompaktní sufixový automat $CDAWG$. Pokud známe při vyhledávání řetězce X indexy A a B, mezi kterými vyhledávat řetězec X ve vstupním řetězci S, lze v konstantním čase určit automaty, které je potřeba prohledat. Dále vznikly programy pro kompresi a dekompresi vstupních dat pomocí 2 bitového kódování. Výsledkem práce jsou experimenty nad touto implementací, které jsou podrobně popsány v předchozí kapitole.

Cílem experimentů bylo nalézt takovou délku rozdělení na podřetězce, při které bylo vyhledávání řetězců nejrychlejší a zároveň bylo úsporné co se týče místa na disku. Proběhlo několik měření rychlosti vyhledání řetězců v dlouhém vstupním řetězci *data64mb* obsahující 64 miliónů znaků. Bylo nalezeno, že ideální délka rozdělení je mezi rozdělením na 2^{17} a 2^{18} znaků kvůli nejrychlejšímu vyhledávání. Pokud je žádoucí zvolit rozdělení, které je efektivnější pro zápis na disk, lze zvolit nejdelší rozdělení pro uložení na 2 byty, které implementace dovoluje, to je 2^{15} znaků. Ušetříme takto zhruba 50% místa na disku v porovnání s ukládáním na 4 byty za cenu zanedbatelně nižšího výkonu při samotném vyhledávání. Na datech *data*64*mb* byly dodatečně vygenerovány další 2 sady dotazů, naměřené časy se příliš nelišily od původních výsledků vyhledávání. Nejefektivnější zápis na 1 byte, tedy při rozdělení 2^7 znaků, se v tomto případě prokázal jako nevhodný z důvodu zhruba 40x pomalejšího vyhledávání oproti zápisu na 2 byty, tedy při rozdělení délky 2^{15} znaků.

Dále bylo dokázáno, že se opravdu vyplatí rozdělovat vstupní řetězec. Byl zkonstruován pouze jeden $CDAWG$ nad celým řetězcem *data64mb*, kde čas konstrukce a rychlost vyhledávání všech výskytů řetězce je více než 25x pomalejší, než při rozdělení na podřetězce délky 2^{17}. Velikost jednoho $CDAWG$ na disku je téměř stejná při srovnání s rozdělením na 2^{17} znaků. Při rozdělení na 2^{15} znaků lze navíc zapsat automaty na disk s využitím poloviny paměti. V případě, kdy se hledá pouze první výskyt řetězce, je vyhledávání nejrychlejší pouze v jednom automatu. Avšak nad velice dlouhým vstupem, jako jsou data *human*, nelze zkonstruovat pouze jeden automat kvůli vysokým nárokům na paměť.

Nedostatkem v implementaci je neoptimálně navržená datová struktura jednoho $CDAWG$, kde každý objekt $CDAWG$ obsahuje příliš mnoho pomocných proměnných, které mohla mít v režii jiná entita. Zároveň byla použita stejná struktura $CDAWG$ pro konstrukci i pro vyhledávání, kde při konstrukci bylo potřeba dodatečných proměnných, které při vyhledávání jsou zbytečné a způsobují vyšší využití paměti za běhu. Díky tomuto nedostatku nebylo možné změřit vyhledávání nad daty *human* bez využití *lazy-loading*.

V této práci by se dalo navázat na implementaci paralelního zpracování konstrukce automatů a vyhledávání. Díky rozdělení vstupu napříč několika automaty lze tyto úlohy velice efektivně paralelizovat. Dále by bylo možné zaměřit se na možnosti vyhledávání se statistickou chybou, které rovněž nebylo součástí této práce.

Odkazy

[1] Alberto Apostolico et al. "Parallel construction of a suffix tree with applications". In: *Algorithmica* 3.1-4 (1988), s. 347–365.

[2] Sagar Aryal. *DNA Sequencing - Maxam–Gilbert and Sanger Dideoxy Method*. URL: https://microbenotes.com/dna-sequencing-maxam-gilbert-and-sanger-dideoxy-method/ (cit. 08. 04. 2019).

[3] Marina Barsky et al. "Suffix trees for very large genomic sequences". In: *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM. 2009, s. 1417–1420.

[4] Anselm Blumer, Andrzej Ehrenfeucht a David Haussler. "Average sizes of suffix trees and dawgs". In: *Discrete Applied Mathematics* 24.1-3 (1989), s. 37–45.

[5] Maxime Crochemore a Wojciech Rytter. *Jewels of stringology: text algorithms*. World Scientific, 2003.

[6] Maxime Crochemore a Renaud Vérin. "Direct construction of compact directed acyclic word graphs". In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 1997, s. 116–129.

[7] Sarah E DeWeerdt, Barbara J Culliton a Mary S Gibbs. *What's a genome?* URL: http://www.genomenewsnetwork.org/resources/whats_a_genome/Chp1_1_1.shtml (cit. 08. 04. 2019).

[8] Amol Ghoting a Konstantin Makarychev. "Serial and parallel methods for i/o efficient suffix tree construction". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM. 2009, s. 827–840.

[9] Ramesh Hariharan. "Optimal parallel suffix tree construction". In: *Journal of Computer and System Sciences* 55.1 (1997), s. 44–69.

[10] The Human Genome Management Information System (HGMIS). *About the Human Genome Project: What is the Human Genome Project*. URL: http://www.ornl.gov/sci/techresources/Human_Genome/project/about.shtml (cit. 08. 04. 2019).

[11] The Human Genome Management Information System (HGMIS). *Electropherogram of DNA sequence*. URL: https://www.genome.gov/edkit/bio1a.html (cit. 08. 04. 2019).

[12] Ela Hunt, Malcolm P Atkinson a Robert W Irving. "Database indexing for large DNA and protein sequence collections". In: *The VLDB Journal* 11.3 (2002), s. 256–271.

[13] Shunsuke Inenaga et al. "On-line construction of compact directed acyclic word graphs". In: *Discrete Applied Mathematics* 146.2 (2005), s. 156–179.

[14] Toru Kasai et al. “Linear-time longest-common-prefix computation in suffix arrays and its applications”. In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 2001, s. 181–192.

[15] Juha Kärkkäinen a Peter Sanders. “Simple linear work suffix array construction”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2003, s. 943–955.

[16] Udi Manber a Gene Myers. “Suffix arrays: a new method for on-line string searches”. In: *siam Journal on Computing* 22.5 (1993), s. 935–948.

[17] Essam Mansour et al. “ERA: efficient serial and parallel suffix tree construction for very long strings”. In: *Proceedings of the VLDB Endowment* 5.1 (2011), s. 49–60.

[18] Allan M Maxam a Walter Gilbert. “A new method for sequencing DNA”. In: *Proceedings of the National Academy of Sciences* 74.2 (1977), s. 560–564.

[19] Edward M McCreight. “A space-economical suffix tree construction algorithm”. In: *Journal of the ACM (JACM)* 23.2 (1976), s. 262–272.

[20] Benjarath Phoophakdee a Mohammed J Zaki. “Genome-scale disk-based suffix tree indexing”. In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM. 2007, s. 833–844.

[21] Frederick Sanger, Steven Nicklen a Alan R Coulson. “DNA sequencing with chain-terminating inhibitors”. In: *Proceedings of the national academy of sciences* 74.12 (1977), s. 5463–5467.

[22] Rodger Staden. “A strategy of DNA sequencing employing computer programs”. In: *Nucleic acids research* 6.7 (1979), s. 2601–2610.

[23] Sandeep Tata, Richard A Hankins a Jignesh M Patel. “Practical suffix tree construction”. In: *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment. 2004, s. 36–47.

[24] Yuanyuan Tian et al. “Practical methods for constructing suffix trees”. In: *The VLDB Journal* 14.3 (2005), s. 281–299.

[25] Esko Ukkonen. “Approximate string-matching over suffix trees”. In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 1993, s. 228–242.

[26] Esko Ukkonen. “On-line construction of suffix trees”. In: *Algorithmica* 14.3 (1995), s. 249–260.